Editor/Assembler

Includes the Editor/Assembler *Solid State Software*™ Command Module and two program diskettes. Designed for use with the TI Home Computer, the TI Memory Expansion unit, and the TI Disk Memory System (TI Disk Drive Controller and one to three TI Disk Memory Drives)—all sold separately.

The Editor/Assembler program was developed by the staff of the Texas Instruments Personal Computer Division:

Allen T. Acree, Jr.
Susan Jean Bailey
Sumiko Endo

The Editor/Assembler owner's manual was developed by the staff of the Texas Instruments Learning Center:

Cheryl Watson
Robert E. Whitsitt, II

With Contributions by:

Ira McComic
Henry C. Mishkoff
Jacquelyn Quiram
Jan E. Stevens

TABLE OF CONTENTS

SECTION 1: GENERAL INFORMATION

The Texas Instruments Editor/Assembler Solid State Software™ Command Module and accompanying diskettes allow you to write programs in the powerful assembly language of the TMS9900 microprocessor built into the TI-99/4 and TI-99/4A Home Computers. This assembly language has all of the features expected from an advanced microprocessor, including both byte- and word-oriented commands, auto-incrementing capability, and a variety of addressing modes.

Your Editor/Assembler package contains a command module, two diskettes (labeled Part A and Part B), this manual, overlays for your computer, and a manual for a game or application program. The command module controls the Editor/Assembler and must be inserted in the console to use the features described in this manual. The diskette labeled Part A contains the Editor, the Assembler, TI BASIC support routines, and both source and object code for the Debugger. The diskette labeled Part B contains the SAVE utility, which allows you to save programs in memory image format, and the source and object code for a game or application program that can be used as an example.

The use of assembly language instead of a higher-level language such as BASIC or Pascal has several advantages. The execution of assembly language programs is much faster. In addition, assembly language gives you access to all machine resources, including functions not available from higher-level languages.

Compared to writing programs in machine language, assembly language is much easier. The instructions are mnemonic codes, which are easier to use and remember than the symbols of object code. In addition, you use expressions as operands and may use decimal numbers in expressions and as operands. Further, the use of assembly language relieves you of the tedious task of writing machine language instructions and keeping track of binary machine addresses within the program.

This manual provides details on creating, editing, assembling, and running assembly language programs on the TI Home Computer and includes explanations of the following.

- The use of the Editor.
- All TMS9900 assembly language instructions and pseudo-instructions.
- Assembler output.

- The utilities provided for reading from and writing to VDP RAM.
- The utilities provided for accessing assembly language programs.
- The utilities provided for accessing the Graphics Programming Language subroutines.
- The seven additional TI BASIC subroutines included in the Editor/Assembler Command Module.
- The utilities for communication between assembly language programs and TI BASIC programs.
- The use of the Debugger program.
- The use of sound, color, graphics, and speech from assembly language programs.

The simplest configuration for running the Editor/Assembler requires the TI Home Computer, the TI Memory Expansion unit, the TI Color Monitor (or the TI Video Modulator and a television set), the Editor/Assembler Command Module, the Editor/Assembler diskette, and a TI Disk Memory System with at least one Disk Memory Drive. With this equipment you can either develop programs of your own or run existing assembly language programs. To enhance your system, you may want to add the RS232 Interface, additional disk drives, or other peripherals available from Texas Instruments.

The Editor in the module allows you to create, edit, print, and save files. Several commands make file preparation as simple as possible. After a program file has been created, you can assemble it with the Assembler in the module.

The Assembler reads a source file prepared in the Editor and produces object code in one of two formats and error messages. It can list the assembled program.

After a file has been assembled, you may load and run it. The Debugger program can help you to find and correct any errors which may occur. When the program is satisfactory, you can save it with the SAVE utility so that you can easily run it as needed.

1.1 USING THIS MANUAL

This manual assumes that you already know a programming language, preferably an assembly language. If you do not, there are many fine books available which teach the basics of assembly language use. After you know these basics, this manual gives the details of TMS9900 assembly language and its application to the TI Home Computer.

When terms that may be new to you are first used, they are defined. (If you want to review a definition later, a Glossary is provided near the end of the manual.) Section 2 explains the basics of using the Editor/Assembler. Sections 3 through 15 are a detailed description of the TMS9900 assembly language and assembler output. The remainder of the manual describes applications specific to the TI Home Computer, such as access to utilities, BASIC support, file management, the linking loader, the debugger utility, and the use of sound, color, graphics (including sprites), and speech.

Several appendices provide other useful information, including a description of the number bases used, the character sets available, the instructions, and related information.

1.2 SET-UP INSTRUCTIONS

Use your Disk Manager to make backup copies of the diskettes supplied with your Editor/Assembler. You may use those copies for your own use. The originals should be kept in a safe place.

Before you use the Editor/Assembler, the Memory Expansion unit and the TI Disk Memory System must be properly attached to the computer and turned on. See the appropriate owner's manuals for complete set-up instructions.

An automatic reset feature is built into the computer so that when a module is inserted into the console the computer returns to the master title screen. All data or program material you have entered is erased.

CAUTION

To avoid damaging the module, be sure it is free of static electricity before inserting it into the computer. Touch any metal object, such as a door knob or desk lamp, before handling the module. Keep the module clean and dry, and do not touch the recessed contacts.

1. Slide the command module into the slot on the console.

2. Turn on all peripherals. Turn on the computer.

3. Insert the diskette labeled Part A into Disk Drive 1.

4. Press any key to make the master selection list appear. To select the module, press the key corresponding to the number beside EDITOR/ASSEMBLER.

Note: If the diskette is not inserted prior to using a function requiring the diskette, you may have to turn the computer off, insert the diskette, and start over.

To remove the module, first return the computer to the master title screen by pressing <quit>. Then remove the module from the slot. If the module is accidentally removed from the slot while the module contents are being used, the computer may behave eratically. To restore the computer to normal operation, turn the computer console off and wait a few seconds. Then reinsert the module and turn the computer on again.

If you have two or three disk drives, it is best to leave the Editor/Assembler diskette in Disk Drive 1 at all times and put your program diskette in Disk Drive 2 or 3. If you have one drive, then you must either keep the files that you create and edit on the Editor/Assembler diskette or alternate putting the Editor/Assembler diskette and your program diskette in the drive. When you need to load, edit, save, print, or run a file from another diskette, first wait until the necessary portion of the Editor/Assembler has been put into memory from the diskette. Then replace the Editor/Assembler diskette with the one that has your file on it. After your file has been loaded, edited, saved, printed, or run, remove your program diskette and replace it with the Editor/Assembler diskette.

1.2.1 In Case of Difficulty

If the Editor/Assembler does not appear to be operating properly, remove the diskette and return to the master title screen by pressing <esc>. Withdraw the module and remove the diskette from the disk drive. (**Note:** In some instances it may be necessary to turn the computer off, wait several seconds, and then turn it on again before proceeding. Always remove diskettes before turning your computer on or off.) Next align the module with the module opening and reinsert it carefully. Then reinsert the diskette. Now press any key to make the master selection list reappear. Repeat the selection process.

If you have any difficulty with your computer or the Editor/Assembler module, please contact the dealer from whom you purchased the unit and/or module for service directions. Additional information concerning service can be found in the User's Reference Guide.

1.3 SPECIAL KEY FUNCTIONS

On the TI-99/4 console, certain keys are used in combination with the SHIFT keys. On the TI-99/4A console, certain keys are used in combination with the FCTN keys. Pressing any key for more than a moment causes that key to be repeated. **Note:** The Editor/Assembler accepts lower-case letters from the TI-99/4A only in comments and text. For this reason, it is usually best to keep the ALPHA LOCK key pressed down. The {, }, [, and] keys are not available on the TI-99/4.

The following table lists the special keys available when using the Editor/Assembler.

Name	TI-99/4	TI-99/4A	Action
<del character>	SHIFT F	FCTN 1	Deletes a character in the Editor.
<ins character>	SHIFT G	FCTN 2	Inserts a character in the Editor.
<delete line>	SHIFT T	FCTN 3	Deletes a line from the screen.
<roll-up>	SHIFT C	FCTN 4	Displays the next 24 lines of the file.
<next-window>	SHIFT W	FCTN 5	Moves the display to the next window.
<roll-down>	SHIFT V	FCTN 6	Displays the previous 24 lines of the file.
<tab>	SHIFT A	FCTN 7	Moves the cursor to the next tab position.
<insert line>	SHIFT R	FCTN 8	Inserts a line.
<esc>	SHIFT Z	FCTN 9	Returns to the previously displayed screen. In the Editor, enters the command mode.
{		FCTN F	Types the left brace {.
}		FCTN G	Types the right brace }.
[		FCTN R	Types the left bracket [.
]		FCTN T	Types the right bracket].
<left-arrow> or <backspace>	SHIFT S	FCTN S	Moves the cursor to the left one character.
<right-arrow>	SHIFT D	FCTN D	Moves the cursor to the right one character.
<down-arrow>	SHIFT X	FCTN X	Moves the cursor down one line.
<up-arrow>	SHIFT E	FCTN E	Moves the cursor up one line.
<return>	ENTER	ENTER	Tells the computer to accept the information that you type.
<quit>	SHIFT Q	FCTN =	Leaves the Editor/Assembler.

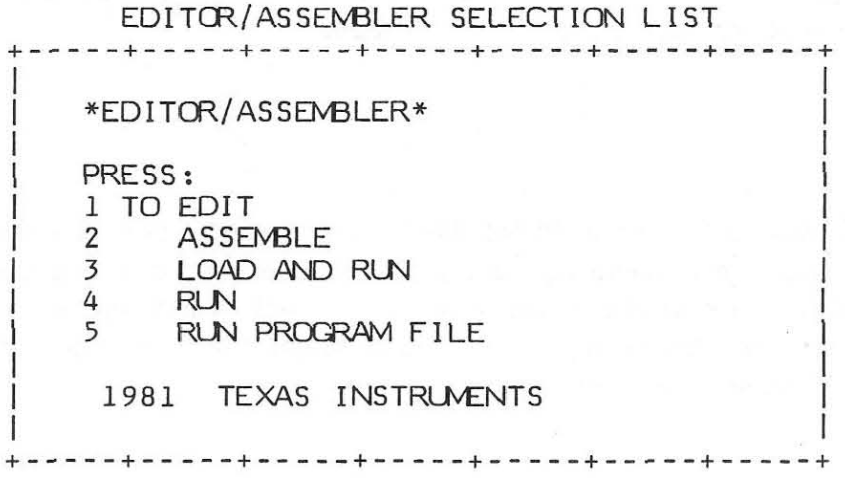
SECTION 2: USING THE EDITOR/ASSEMBLER

This section describes the selections available with the Editor/Assembler. You start the creation of an assembly language program by entering it with the Editor. Then you may save it, load it, and edit it again if necessary. When it is ready, you may assemble it and then load and run it.

The cursor is a flashing marker that appears on the screen to indicate where your next keystroke appears. In editing, the cursor may be moved with the cursor movement keys described in Section 1 or by some of the choices in the command mode of the Editor.

Before using the Editor/Assembler, be certain that all hardware is properly attached and turned on as described in Section 1, with the Editor/Assembler diskette in Disk Drive 1 and the Editor/Assembler module inserted in the console. If you have a TI-99/4A, it is advisable to depress the ALPHA LOCK key.

After you select the module, the Editor/Assembler title appears at the top of the screen, followed by the five options as shown below.



To select an option, press the corresponding number key. At any time you may press <esc> to return to the previous screen or <quit> to return to the master title screen. The five Editor/Assembler options are discussed in the following sections.

2.1 EDITOR

The Editor allows you to load a previously existing file, to create or edit a file, to save a file that you have created or edited, to print a file, or to purge a file from the computer's memory. If you press **1** for EDIT, you enter the Editor mode and the computer displays the following selection list.

EDITOR SELECTION LIST

```
+-----+-----+-----+-----+-----+-----+-----+
|                                           |
|  * EDITOR *                               |
|                                           |
|  PRESS:                                   |
|    1 TO LOAD                              |
|    2    EDIT                              |
|    3    SAVE                              |
|    4    PRINT                             |
|    5    PURGE                             |
|                                           |
+-----+-----+-----+-----+-----+-----+-----+
```

Select LOAD to load an existing file into the computer's memory; EDIT to edit the file in memory; SAVE to save a file from memory; PRINT to print a file from the diskette; or PURGE to delete the file in memory.

2.1.1 Load

A file on a diskette may be loaded for editing or printing. Any file stored in a fixed 80 display format or a variable 80 display format is accepted by the Loader. By saving your files in one of these formats, you may edit a list file or an object file, as well as a source file. However, a compressed object file cannot be edited since it contains undisplayable characters.

Press **1** from the Editor selection list to load an existing file. If the Editor has not already been loaded, the message

ONE MOMENT PLEASE...

is displayed on the screen briefly. Then the prompt

FILE NAME?

appears below the selection list.

If you have a single disk drive, remove the Editor/Assembler diskette and replace it with the diskette that contains the file. With two or three disk drives, place the program diskette in Disk Drive 2 or 3. Type the location and name of the file which you wish to edit, save, or print (such as DSK1.OLDFILE) and press <return>. (You may use the Disk Manager module to obtain a catalog of the files on your diskettes.) The file is located and loaded into memory. The Editor selection list is then displayed and you may select another option. **Note:** Each time a file is loaded, the previous file is removed from memory.

2.1.2 Edit

The Edit option loads the Editor from the Editor/Assembler diskette. The Editor allows you to create a new file or to edit a file which has been loaded with the Load option. When you enter the Editor by pressing **2** from the Editor selection list, the message

ONE MOMENT PLEASE...

is briefly displayed on the screen while the Editor is loaded from the Editor/Assembler diskette. (If the Editor has already been loaded, this message does not appear.) If no file has been loaded, the Edit option clears the screen so that you may begin a new file. The cursor is positioned in the upper left corner of the screen and is followed by the end-of-file marker (*EOF). Press <return> to create a new line. The Editor is now ready to accept your new input.

If a file has been loaded into memory, the Editor displays that file on the screen with the cursor at the top left, ready for you to edit it. You may leave the Editor and return to the Editor selection list by pressing <esc> twice.

Note: The file in memory, whether it is a new file or an existing file, may be lost if you leave the Editor without saving it. Before returning to the Editor/Assembler selection list, be sure to save your program.

The Editor has two modes: the command mode and the edit mode. You are in the edit mode when you first enter the Editor. The command mode is entered from the edit mode by pressing the <esc> key. The edit mode is reentered automatically after you use a command in the command mode.

In the edit mode, the cursor shows where your next keystroke is placed. In the command mode, the cursor is on the second line of the screen ready to accept commands. The command that you enter is effective starting from the position the cursor had when you entered the command mode.

2.1.2.1 *Edit Mode*

In the edit mode, the screen is 80 columns wide with three overlapping 40 column windows available for displaying the text. You start in the left-most window with columns 1 through 40 displayed. Pressing <next-window> moves the display to the center window, with columns 21 through 60 displayed. Pressing <next-window> again moves the display to the right window with columns 41 through 80 displayed. Pressing <next-window> at this point returns the display to the left-most window.

The edit mode allows you to create, modify, and add text to program, data, and text files. When you press a key, that character is placed on the screen in the cursor position and the cursor moves one position to the right. (If the cursor is at the right margin, it moves to the first position on the next line.) In addition, the edit mode has several special keys which perform helpful edit functions. The following table describes the special function keys that are used in the Editor.

Key	Function
<return>	Enters the text into the edit buffer and places the cursor at the start of the next line. If <return> is pressed at the end of the file, a blank line is automatically inserted.
<insert line>	Inserts a blank line above the line where the cursor is located.

Key	Function
<delete line>	Deletes the current line of text, starting at the location of the cursor.
<insert character>	Inserts all characters typed until another function or cursor movement key is pressed. The following characters on the line are moved to the right. The insertion is effective for one line only with all characters after column 80 lost.
<delete character>	Deletes the character under the cursor. The following characters on the line are moved to the left.
<tab>	Moves the cursor right to the next tab location. The tab locations are set at defaults of 1, 8, 13, 26, 31, 46, 60, and 80. To change the tab locations, use the T(AB command in the command mode. If you press <tab> from column 80, the cursor goes to position 1 on the same line.
<next-window>	Moves the cursor position right to the next window so that you may view different portions of the text. If you press <next-window> from the right-most window, the left-most window is displayed.
<roll-up>	Scrolls the screen up by 24-line segments in the edit mode. In the command mode, <roll-up> scrolls the screen by 22-line segments.
<roll-down>	Scrolls the screen down by 24-line segments in the edit mode. In the command mode, <roll-down> scrolls the screen by 22-line segments.
<left-arrow> and <right-arrow>	Allow cursor movement to the left or right without changing the text. When the cursor is at the left margin, pressing <left-arrow> alternately shows and removes the line numbers.
<esc>	Invokes the command mode when in the edit mode. From the command mode, pressing <esc> returns you to the Editor selection list.

The current line number may be displayed and removed by pressing the <left-arrow> key when the cursor is at the left margin. When the line numbers are displayed, the last six characters of the 80-column display cannot be viewed.

2.1.2.2 Command Mode

The command mode, which provides additional editing features, is accessed from the edit mode by pressing <esc>. The command mode uses the first two lines of the screen for promptlines and your input, with the remainder of the screen displaying your file. If an error is detected, the message ERROR appears in the left-hand corner of the second line. Because most of the commands use line numbers, the command mode automatically shows the line numbers. The cursor is displayed on the second line for command input.

The command mode promptline shows the following prompts on a single line at the top of the screen.

E(DIT,F(IND,R(EPLACE,M(OVE,I(NSERT,C(OPY,S(HOW,D(ELETE,A(DJUST,
T(AB,H(OME?

The commands are selected by pressing the first letter of the desired command. All of the edit mode function keys, except <insert-line>, <delete-line>, <up-arrow>, and <down-arrow>, are also effective in the command mode. The <esc> key returns you to the Editor selection list.

The effects of all of the commands except M(OVE, I(NSERT, C(OPY, and D(ELETE start from the position of the cursor when you entered the command mode. The commands E(DIT, A(DJUST, and H(OME occur when you press the letter to choose those commands. The other commands require more information, and occur after that information is entered and <return> is pressed.

The following gives the command mode prompts, in the order in which they appear in the promptline, and describes their functions.

E(DIT — Returns you to the edit mode, with the display as it was before you entered the command mode and the cursor at its previous position.

F(IND Enables you to find a string. The promptline

FIND <CNT>(<COL,COL>)/STRING/

appears on the top line of the screen. You may specify an optional count number, from 1 through 9999, and optional beginning and ending column numbers from 1 through 80.

The count number specifies which occurrence of the string is to be found. If omitted, the default is 1. The two column numbers specify the columns within which the search is to be made. The column numbers must be preceded and followed by parentheses. If the column numbers are omitted, the entire line, columns 1 through 80, is searched. The string must be delimited by slashes (/). The following examples demonstrate the use of F(IND.

Example	Result
/HELLO/	Finds the first occurrence of HELLO.
1000/HELLO/	Finds the 1000th occurrence of HELLO.
(1,50)/ HELLO/	Finds the first occurrence of HELLO in columns 1 through 50.
1000(1,50)/HELLO/	Finds the 1000th occurrence of HELLO in columns 1 through 50.
101/ /	Finds the 101st space.

After the string is located, the Editor leaves the command mode and returns to the edit mode. The string is displayed in line 1 with the cursor located on the first character of the string.

R(EPLACE Replaces the given string with a new string. The promptline

REPLACE<V,><CNT>(<COL,COL>)/OLD/NEW/?

appears at the top of the screen. The count number specifies which occurrence of the string is to be found. If omitted, the default is 1. The two column numbers specify the columns within which the search is to be made. The column numbers must be preceded and followed by

parentheses. If the column numbers are omitted, the entire line, columns 1 through 80, is searched. The old string and new string are entered with slashes delimiting them. After you press <return>, the replacement process begins.

If V (for verify) is specified, the prompt

REPLACE STRING (Y/N/A)

is displayed followed by the string. To replace that occurrence of the string, press **Y**. Press **N** if you do not want to replace the string in that location. The next occurrence of the string is then located (if a count of more than one was specified) and the prompt is again presented. To replace all subsequent occurrences of the specified string, press **A**. The following demonstrate the use of R(EPLACE.

Example	Result
1000/HELLO/GOODBYE/	Changes the first 1000 occurrences of HELLO to GOODBYE.
V,20/HELLO---/BYE/	Presents, one at a time, the first 20 occurrences of HELLO---. You may change them to BYE by pressing **Y**, go on to the next one without changing that one by pressing **N**, or change all subsequent ones by pressing **A**.

M(OVE Displays the promptline

MOVE START LINE, STOP LINE, AFTER LINE?

at the top of the screen. The first value you enter specifies the line number of the beginning of the section to be moved. The second value specifies the line number of the end of the section to be moved. The third value specifies the line after which you want to place the section being moved. For example, if you specify 29 as the AFTER LINE, the data is moved to line 30.

A maximum of a four-digit line number may be specified. However, if the line number is greater than the EOF marker, the line number defaults to the EOF. The EOF line number may be specified by entering E as the starting line, stopping line, or after line. Line number 0 indicates the

line above line number 1. When the move is complete, the line numbers are automatically renumbered. The following examples demonstrate the use of M(OVE.

Example	Result
1,51,57	Moves line 1 through 51 to a position after line 57.
452,E,0	Moves lines 452 through the end of the file to the beginning of the file.

S(HOW Shows the lines starting at the line specified. The promptline

SHOW LINE?

appears. You may respond with a line number or E (to see the line at the end of the file). For example, if you enter 30, the text on line number 30 and all subsequent text is displayed beginning at the top of the screen. The cursor is located on the first character in line number 30.

C(OPY Uses the same promptline and functions in the same manner as M(OVE. However, C(OPY does not delete lines; it places a copy of the designated data at the desired location.

I(NSERT Allows insertion of a file from a diskette before a specified line number. The promptline

INSERT BEFORE LINE, FILE NAME?

requires a line number (four-digit maximum) and the name of the file. For example, 29,DSK2.OLDFILE inserts the file OLDFILE from the diskette in Disk Drive 2 to the file you are editing before line 29.

D(ELETE Deletes the desired text. The text to be deleted is specified as in the M(OVE command. The prompt

DELETE START LINE, STOP LINE?

requires the entry of the beginning and ending line numbers for the deletion.

A(DJUST Returns you to the edit mode. Changes whether numbers are shown. This allows you to see the last six columns of text or data. If the cursor is located in columns 75 through 80, you must first move it to one of the other columns before selecting A(DJUST to leave the line number mode.

T(AB Modifies and sets tabs. The Editor has default tabs at columns 1, 8, 13, 26, 31, 46, 60, and 80. When you choose this command, the top line of the screen displays column numbers (123456789 123456789 ...). The second line has a T located below each of the columns where tab positions are located. Press <space> or <tab> to go to the location where a tab is desired and type T. You may remove tabs by replacing a T with a space. To adjust tabs in columns 75 through 80, tab to column 80 and backspace to the position desired. The tab settings return to the defaults when the Editor is reloaded. **Note:** Do not delete the tab at column 80.

H(OME Moves the cursor to the upper left-hand corner of the screen.

2.1.3 Save

After you have edited a file, it must be saved on diskette for future use. Otherwise, when you leave the Editor, the file may be lost. You save a file by pressing **3** from the Editor selection list. The prompt

VARIABLE 80 FORMAT (Y/N)?

appears at the bottom of the screen. If **Y** is pressed, a file is opened with a variable 80 display format, which uses less space on the diskette than a fixed 80 display format. If you press **N**, a fixed 80 display format is used. After a file is saved on diskette, its format cannot be changed unless the file is reloaded into memory and saved again in the new format.

After the format is chosen, the prompt

FILE NAME?

is displayed. If you have a single disk drive, remove the Editor/Assembler diskette and replace it with the diskette that contains the file. If you have two or three disk drives, place the program diskette in Disk Drive 2 or 3. To save your file on a

diskette, enter the device and filename. For example, DSK1.SAVEFILE saves your file on the diskette in Disk Drive 1 under the name SAVEFILE. After you save your file, be sure that the Editor/Assembler diskette is in Disk Drive 1.

You may also save your file to the RS232 by specifying RS232 as the filename. The output is then directed to the device connected to the RS232, which is normally a printer. When outputting to the RS232, you must specify a file that is in variable 80 format.

You may wish to use the print option to print a file instead of the save option. However, outputting is faster with SAVE than with the PRINT option. The TI Thermal Printer may not be used with the save option. It is only accessible from the print option.

2.1.4 Print

The print option allows you to print a file to the RS232 Interface, the Thermal Printer, or a diskette file. A source, list, object, or any other file in either a variable 80 display format or a fixed 80 display format can be printed. A printed compressed object file may appear somewhat confusing because it contains unprintable characters. Select the print option by pressing **4** on the Editor selection list. The prompt

FILE NAME?

appears on the screen. If you have a single disk drive, remove the Editor/Assembler diskette and replace it with the diskette that contains the file. With two or three disk drives, place the program diskette in Disk Drive 2 or 3. Enter a filename, such as DSK1.OLDFILE. The file must be on a diskette. After you enter the filename, the prompt

DEVICE NAME?

appears. A diskette file, RS232, or TP may be specified as a device name. If the diskette is specified (to duplicate a file) the entire diskette filename, such as DSK1.PRNTFILE, must be entered. The output file is in variable 80 format, so an object file duplicated with the print option cannot be loaded by the Loader.

After you specify the device, the file is printed on that device. The print option does not require that the Editor be in memory. If the Editor is in memory, the print

option does not alter the text being edited, so you may continue to edit after you use the print option.

After you have printed your file, be sure that the Editor/Assembler diskette is in Disk Drive 1.

2.1.5 Purge

The purge option allows you to remove the file currently in memory. After you select the purge option by pressing **5** from the Editor selection list, the prompt

ARE YOU SURE (Y/N)?

appears at the bottom of the screen. If you press **Y**, the file is cleared from memory and is no longer accessible. If you press **N**, the file remains in memory and you are returned to the Editor selection list. You should normally save your file prior to purging it from memory. You normally only purge a file when you wish to create a new file.

2.2 ASSEMBLE

The Assembler allows you to assemble files that you have created, edited, and saved with the Editor. If you press **2** for ASSEMBLE from the Editor/Assembler selection list, you enter the Assembler.

2.2.1 File and Option Specification

If the Assembler has not been previously loaded, the prompt

LOAD ASSEMBLER?

appears on the screen. If you press **N** you are returned to the Editor/Assembler selection list. If you press **Y**, and the Editor/Assembler diskette is in Disk Drive 1, the message ONE MOMENT PLEASE... is displayed and the Assembler is loaded. If the Assembler has already been loaded, the prompt LOAD ASSEMBLER? is omitted.

After the Assembler is loaded, the prompt

SOURCE FILE NAME?

appears. Type the file name, such as DSK1.SOURCE. The source file must be on a diskette in either a fixed or variable 80 display format. Then the prompt

OBJECT FILE NAME?

appears. Type the object file name, such as DSK1.OBJECT. The object file (which is created if it does not already exist) must be on a diskette. It is created by the Assembler in a fixed 80 format. Then the prompt

LIST FILE NAME?

appears. Type the list file name, such as DSK1.LIST. Just press <return> if you do not want a listing. The list file can be output to a diskette file or the RS232 Interface and is always in a variable 80 display format. Then the prompt

OPTIONS?

appears. The options available and their functions are listed on the next page.

Option	Function
R	Defines the Workspace Register symbols R0 through R15 equal to 0 through 15.
L	Specifies list file generation.
S	Specifies that a symbol table dump is to be included in the list file.
C	Specifies that the object file is to be in compressed format to save space on the diskette.

If no option is desired, simply press <return>. If you want more than one option, enter each letter with no commas or spaces between the letters. If L (list file generation) is not specified in the option input, the Assembler does not create a listing even though a list file name was specified. If a list file is not specified, the Assembler assembles the program file more quickly. The R option is almost always required to generate proper object code. If a letter other than L, S, C, or R is specified, it is ignored.

For example, the following shows the prompts and your responses if you wish to assemble a file named SOURCE, list it to the RS232 at 9600 baud, name the object file OBJECT, and use the options L, S, and R.

Prompt	Your input
SOURCE FILE NAME?	DSK1.SOURCE
OBJECT FILE NAME?	DSK1.OBJECT
LIST FILE NAME?	RS232.BA=9600
OPTIONS?	RLS

2.2.2 Assembly

After you enter the options, the program transfers control to the Assembler. While the Assembler is running, the message

ASSEMBLER EXECUTING

appears at the bottom of the screen. If a fatal error is encountered, the Assembler returns the error code and stops assembling. If a non-fatal error is detected, the line number and appropriate error message are displayed at the bottom of the screen and assembly continues. See Section 15 for a complete description of the Assembler output.

When assembly is complete, the total number of errors is shown on the screen, as well as the message

PRESS ENTER TO CONTINUE.

When <return> is pressed, the program returns to the Editor/Assembler selection list.

2.3 LOAD AND RUN

You may load and run the object code produced by the Assembler by pressing 3 for LOAD AND RUN from the Editor/Assembler selection list. When you select LOAD AND RUN, the prompt

FILE NAME?

appears. If you have a single disk drive, remove the Editor/Assembler diskette and replace it with the diskette that contains the file. With two or three disk drives, place the program diskette in Disk Drive 2 or 3. The file must be an object file on a diskette in either regular or compressed object format. Type your filename (such as DSK1.OLDOBJ) and press <return>.

After the file is loaded, the filename is erased from the screen and you may enter another filename. You may load as many files as you like until the memory is full. **Note:** If an error occurs in loading any file, then all files must be loaded again. After you have loaded all your files, you may proceed by pressing <return> without entering a file name.

The prompt

PROGRAM NAME?

appears next. The program name is any entry point in your program marked by a label which has been defined in the DEF list of the program. If you press <return> without entering a program name, the program most recently executed is located and executed.

If the program has an entry label with an END statement, the Loader starts executing from that label without prompting for the program name. If you attempt to run a program in which there are unresolved references, an error occurs.

Note: Once your program has started to run, it is in total control of the computer. Unless the program allows you to return control to the Editor/Assembler, to TI BASIC, or to the master selection list, the only way to stop the program is to turn off the computer.

2.4 RUN

You may run a program that has already been loaded into memory by pressing **4** for RUN from the Editor/Assembler selection list. When you select RUN, the prompt

PROGRAM NAME?

appears. The program name is any entry point in your program marked by a label which has been defined in the DEF list of the program. If you press <return> without entering a program name, the program most recently executed is located and executed.

If the program has an entry label with an END statement, the Loader starts executing from that label without prompting for the program name. If you attempt to run a program in which there are unresolved references, an error occurs.

Note: Once your program has started to run, it is in total control of the computer. Unless the program allows you to return control to the Editor/Assembler, to TI BASIC, or to the master selection list, the only way to stop the program is to turn off the computer.

2.5 RUN PROGRAM FILE

You may load and run a file that is on a diskette or cassette as a memory image file by pressing **5** for RUN PROGRAM FILE from the Editor/Assembler selection list. You may create and save a file as a memory image by using the SAVE utility provided on the second Editor/Assembler diskette. See Section 24.5 for a description of this utility.

Some arcade games are provided by Texas Instruments in this format and may be run using this option. The game on the second Editor/Assembler diskette must be put in this format with the SAVE utility before you can use it.

When you choose this option, the prompt

PROGRAM FILE NAME?

is displayed. Enter the name of the program preceded by the device name. For example, DSK1.GAME is a proper program filename. The program is then loaded and run.

Note: Once your program has started to run, it is in total control of the computer. Unless the program allows you to return control to the Editor/Assembler, to TI BASIC, or to the master selection list, the only way to stop the program is to turn off the computer.

SECTION 3: GENERAL PROGRAMMING INFORMATION

This section discusses how the TI Home Computer and the TMS9900 microprocessor allow you to use Registers, transfer vectors, Workspaces, source statement formats, expressions, constants, symbols, terms, and character strings.

3.1 REGISTERS

A register is a memory word that serves a secific purpose. Registers in Random Access Memory (RAM) are called "software" registers. A set of 16 consecutive registers is called a "workspace."

Three "hardware" registers are located in the CPU itself. They are the Program Counter Register, the Workspace Pointer Register, and the Status Register.

3.1.1 Program Counter Register (PC)

The Program Counter Register (PC) keeps track of the location of the next instruction in memory. The PC manages the program and maintains a sequential and orderly flow of instructions.

3.1.2 Workspace Pointer Register (WP)

The Workspace Pointer Register (WP) contains the address of the current software workspace.

3.1.3 Status Register (ST)

The Status Register (ST) contains indications of the present status of the computer. Each bit of the status register is initialized to zero when the computer is turned on. Then, as each instruction is performed, the computer indicates the status by changing the appropriate "switches" as a result of that instruction. By this method the bits are set (changed to 1) and reset (changed to 0) by machine instructions. Status bits have the following meanings.

Name	Bit Number	Meaning
L>	0	Logical greater than
A>	1	Arithmetic greater than
EQ	2	Equal
C	3	Carry
OV	4	Overflow
OP	5	Odd parity
X	6	Extended operation
-	7-11	Reserved
INT. MASK	12-15	Interrupt mask

In the diagrams in this manual, bits that are checked or set have a caret (^) printed under them. The following is a representation of the Status Register with the L> and EQ bits set.

Status Register

```
 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
|L>|A>|EQ|C |OV|OP|X |--------------| INT. MASK |
   ^     ^
```

The following table indicates the bits in the Status Register that may be affected by the various assembly language instructions.

Status Bits Affected by Instructions[1]

Mnemonic	L>	A>	EQ	C	OV	OP	X
A	X	X	X	X	X	-	-
AB	X	X	X	X	X	X	-
ABS	X	X	X	X	X	-	-
AI	X	X	X	X	X	-	-
ANDI	X	X	X	-	-	-	-
B	-	-	-	-	-	-	-
BL	-	-	-	-	-	-	-
BLWP	-	-	-	-	-	-	-
C	X	X	X	-	-	-	-
CB	X	X	X	-	-	X	-
CI	X	X	X	-	-	-	-
CLR	-	-	-	-	-	-	-
COC	-	-	X	-	-	-	-
CZC	-	-	X	-	-	-	-
DEC	X	X	X	X	X	-	-
DECT	X	X	X	X	X	-	-
DIV	-	-	-	-	X	-	-
INC	X	X	X	X	X	-	-
INCT	X	X	X	X	X	-	-
INV	X	X	X	-	-	-	-
JEQ	-	-	-	-	-	-	-
JGT	-	-	-	-	-	-	-
JH	-	-	-	-	-	-	-
JHE	-	-	-	-	-	-	-
JL	-	-	-	-	-	-	-
JLE	-	-	-	-	-	-	-
JLT	-	-	-	-	-	-	-
JMP	-	-	-	-	-	-	-
JNC	-	-	-	-	-	-	-
JNE	-	-	-	-	-	-	-
JNO	-	-	-	-	-	-	-
JOC	-	-	-	-	-	-	-

Mnemonic	L>	A>	EQ	C	OV	OP	X
JOP	-	-	-	-	-	-	-
LDCR	X	X	X	X	-	2	-
LI	X	X	X	-	-	-	-
LIMI	-	-	-	-	-	-	-
LWPI	-	-	-	-	-	-	-
MOV	X	X	X	-	-	-	-
MOVB	X	X	X	-	-	X	-
MPY	-	-	-	-	-	-	-
NEG	X	X	X	X	X	-	-
ORI	X	X	X	-	-	-	-
RTWP	X	X	X	X	X	X	X
S	X	X	X	X	X	-	-
SB	X	X	X	X	X	X	-
SBO	-	-	-	-	-	-	-
SBZ	-	-	-	-	-	-	-
SETO	-	-	-	-	-	-	-
SLA	X	X	X	X	X	-	-
SOC	X	X	X	-	-	-	-
SOCB	X	X	X	-	-	X	-
SRA	X	X	X	X	-	-	-
SRC	X	X	X	X	-	-	-
SRL	X	X	X	X	-	-	-
STCR	X	X	X	-	-	2	-
STST	-	-	-	-	-	-	-
STWP	-	-	-	-	-	-	-
SWPB	-	-	-	-	-	-	-
SZC	X	X	X	-	-	-	-
SZCB	X	X	X	-	-	X	-
TB	-	-	X	-	-	-	-
X	3	3	3	3	3	3	3
XOP	3	3	3	3	3	3	3
XOR	X	X	X	-	-	-	-

Notes:

[1]In addition to these instructions, the instructions CKOF, CKON, IDLE, LREX, and RSET are included in this manual for completeness. None affect any status bits or have any other useful effect on the Home Computer.

[2]When an LDCR or STCR instruction transfers eight or fewer bits, the OP bit is set or reset as in byte instructions. Otherwise, the OP bit is not affected.

[3]The X instruction does not affect any status bit. The instruction executed by the X instruction sets status bits normally. When an XOP instruction is implemented, the XOP bit is set, and the subroutine sets status bits normally.

3.1.3.1 Logical Greater Than--(L>)

The logical greater than bit is set when an unsigned number is compared with a smaller unsigned number. In this comparison, the most significant bits of the words being compared represent 2^{15}. The least significant bits of the bytes being compared represent 2^{7}.

3.1.3.2 Arithmetic Greater Than--(A>)

The arithmetic greater than bit is set when a signed number is compared with a smaller signed number. The most significant bits of the words or bytes being compared represent the sign of the number, zero for positive or one for negative. For positive numbers, the remaining bits represent the binary value. For negative numbers, the remaining bits represent the two's complement of the binary value.

3.1.3.3 Equal--(EQ)

The equal bit is set when the two words or bytes being compared are equal. The significance of equality is the same whether the comparison is between unsigned binary numbers or two's complement numbers.

3.1.3.4 Carry--(C)

The carry bit is set by a carry of 1 from the most significant bit (sign bit) of a word or byte during arithmetic and shift operations. Thus the carry bit is used by shift operations to store the last bit shifted out of the Workspace Register being shifted.

3.1.3.5 Overflow--(OV)

The overflow bit is set when the result of an arithmetic operation is too large or too small to be represented correctly in two's complement representation.

In addition operations, the overflow bit is set when the most significant bits of the operands are equal and the most significant bit of the result is not equal to the most significant bit of the destination operand.

In subtraction operations, the overflow bit is set when the most significant bits of the operands are not equal and the most significant bit of the result is not equal to the most significant bit of the destination operand.

For a divide operation, the overflow bit is set when the most significant 16 bits of the dividend are greater than or equal to the divisor.

For an arithmetic left shift, the overflow bit is set if the most significant bit of the workspace register being shifted changes value.

For the absolute value and negate instructions, the overflow bit is set when the source operand is the maximum negative value (>8000).

3.1.3.6 Odd Parity--(OP)

In byte operations the odd parity bit is set when the parity of the result is odd and is reset when the parity is even. The parity of a byte is odd when the number of bits having values of one is odd. When the number of bits having values of one is even, the parity of the byte is even. The odd parity bit is equal to the least significant bit of the sum of the bits in the byte.

3.1.3.7 Extended Operation--(X)

The extended operation instruction (XOP) is available in some TI-99/4A computers. The only way to determine if your computer supports this instruction is to try it. Extended operation instructions permit a limited extension of the existing instruction set to include additional instructions. In the computer, these additional instructions are implemented by software routines.

When the program contains an XOP instruction (see Section 14.5) that is software implemented, the computer locates the XOP Workspace Pointer (WP) and Program Counter (PC) words in the XOP reserved memory locations and loads the WP and PC. Then the computer transfers control to the XOP instruction set through a context switch (See Section 3.2). When the context switch is complete, the XOP workspace contains the calling routine's return data in Workspace Registers 13, 14, and 15.

The extended operation bit is set when the software implemented extended operation is initiated.

3.1.3.8 Interrupt Mask

The interrupt mask is status bits 12 through 15. Any device with a level number less than or equal to the value in the interrupt mask is permitted by the TMS9900 microprocessor to interrupt a running program. Thus if the interrupt mask has a value of 2 (binary 0010), any device with a level of 0, 1, or 2 may interrupt a running program. On the TI Home Computer, all interrupts are on level 2. Thus only values of 0 and 2 are useful.

3.2 TRANSFER VECTORS AND WORKSPACE

A transfer vector is two consecutive words of memory which contain a pair of memory addresses. The first word contains the address of a 16-word area of memory called a workspace, and the second word contains the address of a subroutine entry point. The computer uses a transfer vector to perform a transfer of control called a context switch.

A context switch places the contents of the first word of a transfer vector in the Workspace Pointer Register. The active workspace becomes the workspace addressed by that word, with the 16 words of the active workspace called registers 0 through 15. These are available for use as general purpose registers, address registers, or index registers. The context switch places the contents of the second word of a transfer vector in the Program Counter, causing the instruction at that address to be executed next.

3.3 SOURCE STATEMENT FORMAT

An assembly language source program consists of source statements which may contain assembler directives, machine instructions, pseudo-instructions, or comments.

Each line (or record) of a source statement consists of a maximum of 80 characters of information (including spaces). A record may be subdivided into several variably sized sections known as fields.

The label field is positioned at the beginning of the source statement and serves as a reference point. The op-code field is the operation code (a number, name, or abbreviation) of the task to be performed by that source statement. The operand field stipulates the value that is to be operated upon or manipulated. It may be a number, string, address, etc. The comment field is an area reserved for you to make comments that increase the readability of the program but that do not affect the operations of the computer. The syntax definition describes the required form for the use of commands as related to the fields. Section 4 describes formatting procedures and definitions in detail.

The following conventions apply in the syntax definitions for machine instructions and assembler directives.

- Items in capital letters, including special characters, must be entered exactly as shown.
- Items within angle brackets (<>) are defined by you.
- Items in lower-case letters represent classes (generic names) of items.
- Items within brackets ([]) are optional.
- Items within braces ({ }) are alternative items, one of which must be entered.
- An ellipsis (...) indicates that the preceding item may be repeated.
- The symbol b represents one or more blanks or spaces.

The syntax (required form) for source statements other than comment statements is defined as follows.

[<label>] b op-code b [<operand>] [,<operand>] ... b [<comment>]

As this syntax definition indicates, a source statement may have a label, which you define. One or more blanks separate the label from the op-code. Mnemonic operation codes, assembler directive codes, and pseudo-operation codes are all included in the generic term op-code, and you may enter any of these. One or more blanks separate the op-code from the operand, when an operand is required.

Additional operands, when required, are separated by commas. One or more blanks separate the operand or operands from the comment field.

Note: Although the maximum length of a source record is 80 characters, the list file displays only the first 60 characters of each line.

3.3.1 Character Set

The Assembler recognizes the following ASCII characters.

The alphabet (upper-case letters only except in comment and text fields) and space character
The numerals 0 through 9
Several special characters and control characters

The character set is listed in the Appendix.

3.3.2 Label Field

The label field begins in the first character position of the source record and extends to the first blank. The label field consists of a symbol containing up to six characters, the first of which must be alphabetic. Additional characters may be any alphanumeric characters. A label is optional for machine instructions and for many assembler directives. When the label is omitted, however, the first character position must contain a blank.

A source statement consisting of only a label field is a valid statement. It has the effect of assigning the current location to the label. This is usually equivalent to placing the label in the label field of the following machine instruction or assembler directive. However, when a statement consisting of only a label is preceded by a TEXT or BYTE directive and is followed by a DATA directive or a machine instruction, the label does not have the same value as a label in the following statement unless the TEXT or BYTE directive left the location counter on an even (word) location. An EVEN directive following the TEXT or BYTE directive prevents this problem.

3.3.3 Operation Field

The operation (op-code) field begins after the blank that terminates the label field or in the first non-blank character position after the first character position when the label is omitted. The operation field is terminated by one or more blanks and may not extend past character position 60 of the source record. The operation field contains an op-code, which is one of the following.

- Mnemonic operation code of a machine instruction
- Assembler directive operation code
- Symbol assigned to an extended operation by a DXOP directive
- Pseudo-instruction operation code

3.3.4 Operand Field

The operand field begins after the blank that terminates the operation field. It may not extend past character position 60 of the source record. The operand field may contain one or more expressions, terms, or constants, according to the requirements of the particular op-code. The operand field is terminated by one or more blanks.

3.3.5 Comment Field and Comment Line

The comment field begins after the blank that terminates the operand field, and may extend to the end of the source record if required. The comment field may contain any ASCII character, including blank. The contents of the comment field are listed in the source portion of the assembly listing but have no other effect on the assembly.

Comment statements consist of a single field starting with an asterisk (*) in the first character position and followed by any ASCII character, including a blank, in each succeeding character position. Comment statements are listed in the source portion of the assembly listing, but have no other effect on the assembly. A totally blank line is also treated as a comment line.

3.4 EXPRESSIONS

Expressions are used in the operand fields of assembler directives and machine instructions. An expression is a constant or symbol, a series of constants or symbols, or a series of constants and symbols separated by arithmetic operators. Each constant or symbol may be preceded by a minus sign (unary minus) to indicate a negative value. An expression may contain no embedded blanks or symbols that are defined as extended operations. Symbols that are defined as external references may not be operands of arithmetic operations. An expression may contain more than one symbol that is not previously defined. When these symbols are absolute, they may also be operands of multiplication or division operations within an expression. The Assembler only supports program-relocatable symbols.

3.4.1 Well-Defined Expressions

Some assembler directives (noted in their descriptions) require well-defined expressions in the operand fields. For an expression to be well-defined, any symbols or assembly-time constants in the expression must have been previously defined. Also, the evaluation of a well-defined expression must be absolute, and a well-defined expression may not contain a character constant.

3.4.2 Arithmetic Operators

The arithmetic operators in expressions are as follows.

+ for addition
- for subtraction
* for multiplication
/ for signed division

In evaluating an expression, the Assembler first negates any constant or symbol preceded by a minus sign (unary minus) and then performs the arithmetic operations from left to right. The Assembler does not assign precedence to any operation other than unary minus. All operations are integer operations. The Assembler truncates the fraction in division.

For example, the expression 4+5*2 is evaluated as 18, not 14, and the expression 7+1/2 is evaluated as 4, not 7. **Note:** Parentheses may not be used to alter the order of the evaluation of expressions.

3.5 CONSTANTS

Constants are used in expressions. The Assembler recognizes four types of constants: decimal integer constants, hexadecimal integer constants, character constants, and assembly-time constants.

3.5.1 Decimal Integer Constants

A decimal integer constant is written as a string of numerals. The range of values of decimal integers is -32,768 to +65,535. Positive decimal integer constants greater than 32,767 are considered negative when interpreted as two's-complement values. Operands of arithmetic instructions other than multiply and divide are interpreted as two's complement numbers, and all comparisons compare numbers both as signed and unsigned values.

The following are valid decimal constants.

1000	Constant, equal to 1,000 or >3E8.
-32768	Constant, equal to -32,768 or >8000.

3.5.2 Hexadecimal Integer Constants

A hexadecimal integer constant is written as a string of up to four hexadecimal numerals preceded by a greater than (>) sign. Hexadecimal numerals include the decimal values 0 through 9 and letters A through F.

The following are valid hexadecimal constants.

>F	Constant, equal to 15, or >F.
>37AC	Constant, equal to 14252 or >37AC.

3.5.3 Character Constants

A character constant is written as a string of one or two characters enclosed in single quotes. To represent a single quote within a character constant, two consecutive single quotes are necessary. The characters are represented internally as eight-bit ASCII characters, with the leading bit set to zero. A character constant consisting only of two single quotes (no character) is valid. This is the null string and is assigned the value >0000.

The following are valid character constants.

'AB'	Represented internally as >4142.
'C'	Represented internally as >43.
'''D'	Represented internally as >2744.

3.5.4 Assembly-Time Constants

An assembly-time constant is written as an expression in the operand field of an EQU directive. (See Section 14.3.) Any symbol in the expression must have been previously defined. The value of the label is determined at assembly time and is considered to be absolute or relocatable according to the relocatability of the expression, not according to the relocatability of the Location Counter value.

3.6 SYMBOLS

Symbols may be used in the label field, the operator field, and the operand field. A symbol is a string of alphanumeric characters, (A through Z and 0 through 9), the first of which must be an alphabetic character, and none of which may be a blank. When more than six characters are used in a symbol, the Assembler prints all the characters but accepts only the first six characters for processing. User-defined symbols are valid only during the assembly in which they are defined.

Symbols used in the label field become symbolic addresses. They are associated with locations in the program and must not be used in the label field of other statements. Mnemonic operation codes and assembler directive names are valid user-defined symbols when placed in the label field.

The DXOP directive defines a symbol to be used in the operator field. Any symbol that is used in the operand field must be placed in the label field of a statement or in the operand field of a REF directive, except for a symbol in the operand field of a DXOP directive or a predefined symbol.

3.7 PREDEFINED SYMBOLS

The predefined symbols are the dollar-sign character ($) and the Workspace Register symbols. The dollar-sign character is used to represent the current location within the program. The 16 Workspace Register symbols are R0 through R15. They are undefined unless you choose the R option when you run the Assembler.

The following are examples of valid symbols.

A1	Assigned the value of the location at which it appears in the label field.
OPERATION	Truncated to the first six letters and assigned the value of the location at which it appears in the label field.
$	Represents the current location.

3.8 TERMS

Terms may be used in the operand fields of machine instructions and assembler directives. A term is a decimal or hexadecimal constant, an absolute assembly-time constant, or a label having an absolute value.

The following are examples of valid terms.

12	Has a value of 12 or >C.
>C	Has a value of 12 or >C.
WR2	Is valid if Workspace Register 2 is defined as having an absolute value.

If START is a relocatable symbol, the following statement is not valid as a term.

WR2	EQU	START+4	WR2 is a relocatable value 4 greater than the value of START. Not valid as a term but valid as a symbol.

3.9 CHARACTER STRINGS

Several assembler directives require character strings in the operand field. A character string is written as a string of characters enclosed in single quotes. To represent a quote within a character string, two consecutive single quotes are necessary. The maximum length of the string is defined by each directive that requires a character string. The characters are represented internally as eight-bit ASCII characters, with the leading bits set to zeros.

The following are valid character strings.

'SAMPLE PROGRAM'	Defines a 14-character string consisting of SAMPLE PROGRAM.
'PLAN "C"'	Defines an 8-character string consisting of PLAN 'C'.
'OPERATOR MESSAGE * PRESS START SWITCH'	Defines a 37-character string consisting of the expression enclosed in single quotes.

SECTION 4: ADDRESSING MODES

This section describes the addressing modes used in assembly language. Examples of programming in each addressing mode are included.

4.1 GENERAL ADDRESSING MODES

A source operand is the number, address, string, etc., which is to be manipulated or operated upon. A destination operand is the address where the result of the performed manipulation is stored. Instructions that specify a general address for a source or destination operand may be in one of five addressing modes. These addressing modes and their uses are discussed in this section.

The following lists the T-field value, which indicates the type of addressing mode (see Section 5), and gives an example for each of the addressing modes.

Addressing Modes

Addressing Mode	T-field value	Example
Workspace Register	00	5
Workspace Register Indirect	01	*7
Symbolic Memory[1, 2]	10	@LABEL
Indexed Memory[1, 3]	10	@LABEL(5)
Workspace Register Indirect Auto-increment	11	*7+

Notes:

[1]The instruction requires an additional word for each T-field value of 10. The additional word contains a memory address.

[2]The four-bit field immediately following the T-field value of 10_2, called the S (for a source operand) or D (for a destination operand) field, is set to zero by the Assembler.

[3]The T-field value of 10_2 indicates both symbolic and indexed memory addressing modes. If the four-bit field which follows it contains a zero value, it is a symbolic addressing mode. If it is non-zero, it is an indexed addressing mode, and the non-zero value is the number of the index register. Therefore, Workspace Register 0 cannot be used for indexing.

4.1.1 Workspace Register Addressing

Workspace Register addressing specifies the Workspace Register that contains the operand. A Workspace Register address is specified by a value of 0 through 15 preceded with an "R". For example, Workspace Register 8 is referred to as "R8".

Examples:

MOV	R4,R8	Copies the contents of Workspace Register 4 into Workspace Register 8.
COC	R15,R10	Compares the bits of Workspace Register 10 that correspond to the one bits in Workspace Register 15 to one.

4.1.2 Workspace Register Indirect Addressing

Workspace Register indirect addressing specifies a Workspace Register that contains the address of the operand. An indirect Workspace Register address is preceded by an asterisk (*).

Examples:

A	*R7,*R2	Adds the contents of the word at the address in Workspace Register 7 to the contents of the word at the address in Workspace Register 2 and places the sum in the word at the address in Workspace Register 2.
MOV	*R7,R0	Copies the contents of the word at the address given in Workspace Register 7 into Workspace Register 0.

4.1.3 Workspace Register *Indirect Auto-Increment* Addressing

Workspace Register indirect auto-increment addressing specifies a Workspace Register that contains the address of the operand. After the address is obtained from the Workspace Register, the Workspace Register is incremented by 1 for a byte instruction or by 2 for a word instruction. A Workspace Register auto-increment address is preceded by an asterisk and followed by a plus sign (+).

Examples:

S	*R3+,R2	Subtracts the contents of the word at the address in Workspace Register 3 from the contents of Workspace Register 2, places the result in Workspace Register 2, and increments the address in Workspace Register 3 by two.
C	R5,*R6+	Compares the contents of Workspace Register 5 with the contents of the word at the address in Workspace Register 6 and increments the address in Workspace Register 6 by two.

4.1.4 Symbolic Memory Addressing

Symbolic memory addressing specifies the memory address that contains the operand. A symbolic memory address is preceded by an "at" sign (@).

Examples:

S	@FIX1,@LIST4	Subtracts the contents of the word at location FIX1 from the contents of the word at location LIST4 and places the difference in the word at location LIST4.
C	R0,@STORE	Compares the contents of Workspace Register 0 with the contents of the word at location STORE.
MOV	@12,@>7C	Copies the word at address >000C into location >007C.

4.1.5 Indexed Memory Addressing

Indexed memory addressing specifies the memory address that contains the operand. The address is the sum of the contents of a Workspace Register and a symbolic address. An indexed memory address is preceded by an "at" sign (@) and followed by a term enclosed in parentheses. The Workspace Register specified by the term within the parentheses is the index register. Workspace Register 0 may not be specified as an index register.

Examples:

A	@2(R7),R6	Adds the contents of the word found at the address computed by adding 2 to the contents of Workspace Register 7 to the contents of Workspace Register 6 and places the sum in Workspace Register 6.
MOV	R7,@LIST4-6(R5)	Copies the contents of Workspace Register 7 into a word of memory. The address of the word of memory is the sum of the contents of Workspace Register 5 and the value of symbol LIST4 minus 6.

4.2 PROGRAM COUNTER RELATIVE ADDRESSING

Program Counter relative addressing is used only by jump instructions. A Program Counter relative address is written as an expression that corresponds to an address at a word boundary. The Assembler evaluates the expression and subtracts the sum of the current location plus two. One-half of the difference is the value placed in the object code. This value must be in the range of -128 through +127. When the instruction is in relocatable code (that is, when the Location Counter is relocatable), the relocation type of the evaluated expression must match the relocation type of the current Location Counter. When the instruction is in absolute code, the expression must be absolute.

Example:

```
JMP     THERE           Jumps unconditionally to location THERE.
```

4.3 CRU BIT ADDRESSING

The CRU, or Communications Register Unit, is a command-driven bit-addressable I/O interface. An instruction can set, reset, or test any bit in the CRU array or move data between the memory and CRU data fields. The CRU software base address is contained in the 16 bits of Workspace Register 12. From the CRU software base address, the processor is able to determine the CRU hardware base address and the resulting CRU bit address.

The CRU bit instructions use a well-defined expression that represents a displacement from the CRU base address (bits 3 through 14). The displacement, in the range of -128 through +127, is added to the base address in Workspace Register 12. See Sections 9 and 24.3 for more information.

Example:

```
SBO     8       Sets CRU bit to one at the CRU address 8 greater
                than the CRU base address.
```

4.3 CRU BIT ADDRESSING

The CRU, or Communications Register Unit, is a command-driven bit-addressable I/O interface. An instruction can set, reset, or test any bit in the CRU array or move data between the memory and CRU data fields. The CRU software base address is contained in the 16 bits of Workspace Register 12. From the CRU software base address, the processor is able to determine the CRU hardware base address and the resulting CRU bit address.

The CRU bit instructions use a well-defined expression that represents a displacement from the CRU base address (bits 3 through 14). The displacement, in the range of -128 through +127, is added to the base address in Workspace Register 12. See Sections 9 and 24.3 for more information.

Example:

```
SBO     8       Sets CRU bit to one at the CRU address 8 greater
                than the CRU base address.
```

4.4 IMMEDIATE ADDRESSING

Immediate instructions use the contents of the word following the instruction word as the operand of the instruction. The immediate value is an expression, and the Assembler places its value in the word following the instruction. Immediate instructions that require two operands have a Workspace Register address preceding the immediate value.

Example:

```
LI      R5,>1000        Places >1000 into Workspace Register 5.
```

4.5 ADDRESSING SUMMARY

The following table shows the addressing mode required for each instruction of the Assembler instruction set. The first column lists the instruction mnemonic. The second and third columns specify the required address, listed below.

G - General address:
  Workspace Register address
  Indirect Workspace Register address
  Symbolic memory address
  Indexed memory address
  Indirect Workspace Register auto-increment address
WR - Workspace Register address
PC - Program counter relative address
CRU - CRU bit address
I - Immediate value
* - The address into which the result is placed when two operands are required

Instruction Addressing

Mnemonic	First Operand	Second Operand	Mnemonic	First Operand	Second Operand
A	G	G*	LDCR	G	Note 1
AB	G	G*	LI	WR*	I
ABS	G	-	LIMI	I	-
AI	WR*	I	LREX	-	-
ANDI	WR*	I	LWPI	I	-
B	G	-	MOV	G	G*
BL	G	-	MOVB	G	G*
BLWP	G	-	MPY	G	WR*
C	G	G	NEG	G	-
CB	G	G	ORI	WR*	I
CI	WR	I	RSET	-	-
CKOF	-	-	RTWP	-	-
CKON	-	-	S	G	G*
CLR	G	-	SB	G	G*
COC	G	WR	SBO	CRU	-
CZC	G	WR	SBZ	CRU	-
DEC	G	-	SETO	G	-
DECT	G	-	SLA	WR*	Note 2
DIV	G	WR*	SOC	G	G*
IDLE	-	-	SOCB	G	G*
INC	G	-	SRA	WR*	Note 2
INCT	G	-	SRC	WR*	Note 2
INV	G	-	SRL	WR*	Note 2

Mnemonic	First Operand	Second Operand
JEQ	PC	-
JGT	PC	-
JH	PC	-
JHE	PC	-
JL	PC	-
JLE	PC	-
JLT	PC	-
JMP	PC	-
JNC	PC	-
JNE	PC	-
JNO	PC	-
JOC	PC	-
JOP	PC	-

Mnemonic	First Operand	Second Operand
STCR	G*	Note 1
STST	WR	-
STWP	WR	-
SWPB	G	-
SZC	G	G*
SZCB	G	G*
TB	CRU	-
X	G	-
XOP	G	Note 3
XOR	G	WR*

Notes:

[1]The second operand is the number of bits to be transferred, from 0 through 15, with 0 meaning 16 bits.

[2]The second operand is the shift count, from 0 through 15. 0 indicates that the count is in bits 12 through 15 of Workspace Register 0. When the count is 0 and bits 12 through 15 of Workspace Register 0 equal 0, the count is 16.

[3]The second operand specifies the extended operation, from 0 through 15. The disposition of the result may or may not be in the first operand address, as determined by you.

SECTION 5: INSTRUCTION FORMATS

An assembler instruction occupies one word (16 bits) of memory. Each word is divided into appropriately sized bit fields which are arranged in one of nine formats. These formats are discussed below and are referred to in the discussions of the instructions in the following sections. You must clearly understand addressing modes, as described in Section 4, before reading this section.

Each format contains one or more of the following bit fields.

Op-Code - Machine operation code.

B - Byte indicator: 1 for byte instructions, 0 for word instructions.

Td - Type of addressing mode of the destination operand.

D - Destination operand.

Ts - Type of addressing mode of the source operand.

S - Source operand.

DISP - Displacement value (signed).

C - Count (bit count).

W - Workspace register.

5.1 FORMAT I -- TWO GENERAL ADDRESS INSTRUCTIONS

The operand field of Format I instructions contains two general addresses separated by a comma. The first address is the source address and the second is the destination address. The Format I mnemonic operation codes are listed below and discussed in subsequent sections.

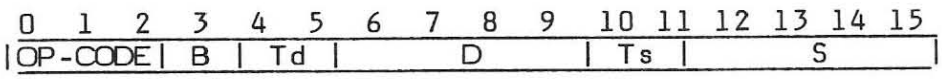
A	Add words
AB	Add Bytes
C	Compare words
CB	Compare Bytes
MOV	MOVe word
MOVB	MOVe Byte
S	Subtract words
SB	Subtract Bytes
SOC	Set Ones Corresponding
SOCB	Set Ones Corresponding, Byte
SZC	Set Zeros Corresponding
SZCB	Set Zeros Corresponding, Byte

Example:

```
SUM     A     @LABEL1,*R7
```

Adds the contents of the word at location LABEL1 to the contents of the word at the address in Workspace Register 7 and places the sum in the word at the address in Workspace Register 7. SUM is the location of the instruction.

Format I instructions are assembled as follows.



When either Ts or Td (but not both) equal binary 10, the instruction occupies two words of memory. The second word contains a memory address used with S or D to develop the effective address. When both Ts and Td equal binary 10, the instruction occupies three words of memory. The second word contains the memory address of the source operand, and the third word contains the memory address of the destination operand.

5.2 FORMAT II -- JUMP INSTRUCTIONS

Format II instructions use Program Counter (PC) relative addresses coded as expressions corresponding to instruction locations on word boundaries. The Format II jump mnemonic operation codes are listed below and discussed in subsequent sections. See Section 5.2.1 for a discussion of the Format II CRU bit I/O instructions.

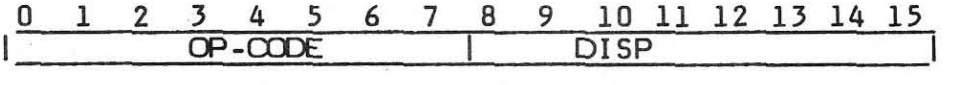
JEQ	Jump if EQual
JGT	Jump if Greater Than
JH	Jump if logical High
JHE	Jump if High or Equal
JL	Jump if logical Low
JLE	Jump if Low or Equal
JLT	Jump if Less Than
JMP	unconditional JuMP
JNC	Jump if No Carry
JNE	Jump if Not Equal
JNO	Jump if No Overflow
JOC	Jump On Carry
JOP	Jump if Odd Parity

Example:

```
NOW     JMP     BEGIN
```

Jumps unconditionally to the instruction at location BEGIN. The address of location BEGIN must not be greater than the address of location NOW by more than 128 words, nor less than the address of location NOW by more than 127 words.

Format II instructions are assembled as follows.



The signed displacement value is shifted one bit position to the left and added to the contents of the Program Counter after the Program Counter has been incremented to the address of the following instruction. In other words, it is a displacement in words from the instruction address plus two.

5.2.1 Format II -- Bit I/O Instructions

In addition to jump instructions, the CRU bit I/O instructions also follow Format II. The operand field of Format II CRU bit I/O instructions contains a well-defined expression which evaluates to a CRU bit address, relative to the contents of Workspace Register 12. The Format II CRU bit I/O instructions are listed below and discussed in subsequent sections. See Section 5.2 for a discussion of the Format II jump instructions.

SBO	Set Bit to logic One
SBZ	Set Bit to logic Zero
TB	Test Bit

Example:

```
SBO     5       Sets a CRU bit to one.
```

5.3 FORMAT III -- LOGICAL INSTRUCTIONS

The operand field of Format III instructions contains a general address followed by a comma and a Workspace Register address. The general address is the source address. The Workspace Register address is the destination address. The Format III mnemonic operation codes are listed below and discussed in subsequent sections.

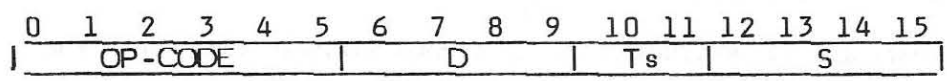
COC	Compare Ones Corresponding
CZC	Compare Zeros Corresponding
XOR	eXclusive OR

Example:

```
COMP    XOR     @LABEL8(R3),R5
```

Performs an exclusive OR operation on the contents of a memory word and the contents of Workspace Register 5 and places the result in Workspace Register 5. The address of the memory word is the sum of the contents of Workspace Register 3 and the value of the symbol LABEL8.

Format III instructions are assembled as follows.



When Ts equals binary 10, the instruction occupies two words of memory. The second word contains the memory address of the source operand.

5.4 FORMAT IV -- CRU MULTI-BIT INSTRUCTIONS

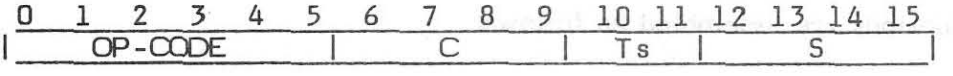
The operand field of Format IV instructions contains a general address followed by a comma and a well-defined expression. The general address is the memory address from which or into which bits are transferred. The CRU address for the transfer is the contents of bits 3 through 14 of Workspace Register 12. The well-defined expression is the number of bits to be transferred and must have a value of 0 through 15. A 0 value specifies a 16 bit transfer. For eight or fewer bits the general address is a byte address. For nine or more bits the general address is a word address. The Format IV mnemonic operation codes are listed below and discussed in subsequent sections.

LDCR	LoaD CRU
STCR	STore CRU

Example:

LDCR	*R6+,8	Places eight bits from the byte of memory at the address in Workspace Register 6 into eight consecutive CRU lines and increments Workspace Register 6 by 1.

Format IV instructions are assembled as follows.



When Ts equals binary 10, the instruction occupies two words of memory. The second word contains the memory address for the source operand.

5.5 FORMAT V -- REGISTER SHIFT INSTRUCTIONS

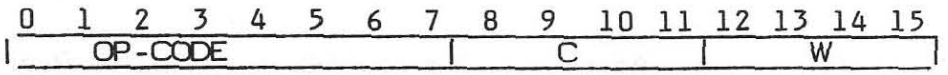
The operand field of Format V instructions contains a Workspace Register address followed by a comma and a well-defined expression. The contents of the Workspace Register are shifted a number of bit positions specified by the well-defined expression. When the term equals zero, the shift count must be placed in bits 12-15 of Workspace Register 0. The Format V mnemonic operation codes are listed below and discussed in subsequent sections.

SLA	Shift Left Arithmetic
SRA	Shift Right Arithmetic
SRC	Shift Right Circular
SRL	Shift Right Logical

Example:

SLA	R6,4	Shifts the contents of Workspace Register 6 to the left 4 bit positions and replaces the vacated bits with zeros.

Format V instructions are assembled as follows.



5.6 FORMAT VI -- SINGLE ADDRESS INSTRUCTIONS

The operand field of Format VI instructions contains a general address. The Format VI mnemonic operation codes are listed below and discussed in subsequent sections.

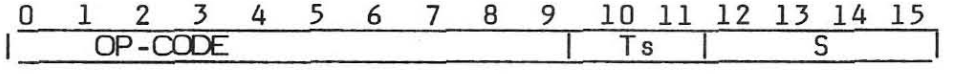
ABS	ABSolute value
B	Branch
BL	Branch and Link
BLWP	Branch and Load Workspace Pointer
CLR	CLeaR
DEC	DECrement
DECT	DECrement by Two
INC	INCrement
INCT	INCrement by Two
INV	INVert
NEG	NEGate
SETO	SEt To One
SWPB	SWaP Bytes
X	eXecute

Example:

```
CNT     INC     R7
```

Adds one to the contents of Workspace Register 7 and places the sum in Workspace Register 7. CNT is the location into which the instruction is placed.

Format VI instructions are assembled as follows.



When Ts equals binary 10, the instruction occupies two words of memory. The second word contains the memory address of the source operand.

5.7 FORMAT VII -- CONTROL INSTRUCTIONS

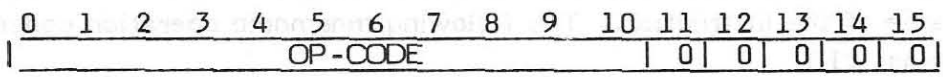
Format VII instructions require no operand field. The Format VII mnemonic operation codes are listed below and discussed in subsequent sections. All but the last instruction have no effect on the TI Home Computer.

CKOF	ClocK OFf
CKON	ClocK ON
IDLE	IDLE
LREX	Load or REstart eXecution
RSET	ReSET
RTWP	ReTurn with Workspace Pointer

Example:

RTWP Returns control to the calling program and restores the context of the calling program by placing the contents of Workspace Registers 13, 14, and 15 into the Workspace Pointer Register, the Program Counter, and the Status Register.

Format VII instructions are assembled as follows.



The op-code field contains 11 bits that define the machine operation. The five least significant bits are zeros.

5.8 FORMAT VIII -- IMMEDIATE INSTRUCTIONS

The operand field of Format VIII instructions contains a Workspace Register address followed by a comma and an expression. The Workspace Register is the destination address, and the expression is the immediate operand. The Format VIII mnemonic operation codes are listed below and discussed in subsequent sections.

AI	Add Immediate
ANDI	AND Immediate
CI	Compare Immediate
LI	Load Immediate
ORI	OR Immediate

There are two additional Format VIII instructions that require only an expression in the operand field. The expression is the immediate operand. The destination is implied in the name of the instruction. These instructions are listed here.

LIMI	Load Interupt Mask Immediate
LWPI	Load Workspace Pointer Immediate

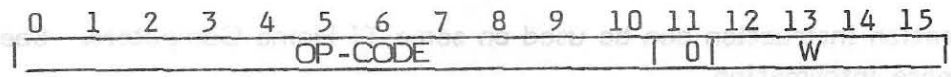
Another modification of Format VIII requires only a Workspace Register address in the operand field. The Workspace Register address is the destination. The source is implied in the name of the instruction. The following mnemonic operation codes use this modified Format VIII.

STST	STore STatus
STWP	STore Workspace Pointer

Examples:

ANDI	4,>000F	Performs an AND operation on the contents of Workspace Register 4 and immediate operand >000F.
LWPI	WRK1	Places the address defined for the symbol WRK1 into the Workspace Pointer Register.
STWP	R4	Places the contents of the Workspace Pointer Register into Workspace Register 4.

Format VIII instructions are assembled as follows.



A zero bit separates the two fields. The instructions that have no Workspace Register operand place zeros in the W field. The instructions that have immediate operands place the operands in the word following the word that contains the op-code, i.e., these instructions occupy two words each.

5.9 FORMAT IX -- EXTENDED OPERATION INSTRUCTION

The extended operation instruction can be used on some TI Home Computers. See Section 7.19 for more information.

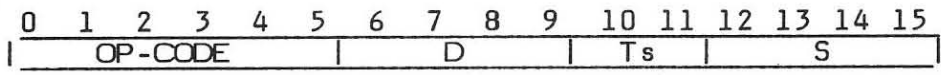
The operand field of the Format IX extended operation instruction contains a general address and a well-defined expression. The general address is the address of the operand for the extended operation. The term specifies the extended operation to be performed and must be in the range of 0 through 15. The Format IX mnemonic operation code is listed below and discussed in subsequent sections. See Section 5.9.1 for a discussion of the Format IX multiply and divide instructions.

XOP	eXtended OPeration

Example:

XOP	@LABEL(R4),12	Performs extended operation 12 using the address computed by adding the value of symbol LABEL to the contents of Workspace Register 4.

Format IX instructions are assembled as follows.



When Ts equals binary 10, the instruction occupies two words of memory. The second word contains the memory address for the source operand.

5.9.1 Format IX -- Multiply and Divide Instructions

The operand field of Format IX multiply and divide instructions contains a general address followed by a comma and a Workspace Register address. The general address is the address of the multiplier or divisor, and the Workspace Register address is the address of the Workspace Register that contains the multiplicand or dividend. The Workspace Register address is also the address of the first of two Workspace Registers to contain the result. The Format IX multiply and divide instructions are listed below and discussed in subsequent sections. See Section 5.9 for a discussion of the Format IX extended operation instruction.

MPY	MultiPlY
DIV	DIVide

Example:

MPY	@ACC,R9	Multiplies the contents of Workspace Register 9 by the contents of the word at location ACC and places the product in Workspace Registers 9 and 10, with the 16 least significant bits of the product in Workspace Register 10.

Multiply and divide instructions are assembled in the same format as shown in Section 5.9, except that the D field contains the Workspace Register operand.

SECTION 6: ARITHMETIC INSTRUCTIONS

The following arithmetic instructions are described in this section.

Instruction	Mnemonic	Section
Add words	A	6.1
Add Bytes	AB	6.2
ABSolute value	ABS	6.3
Add Immediate	AI	6.4
DECrement	DEC	6.5
DECrement by Two	DECT	6.6
DIVide	DIV	6.7
INCrement	INC	6.8
INCrement by Two	INCT	6.9
MultiPlY	MPY	6.10
NEGate	NEG	6.11
Subtract words	S	6.12
Subtract Bytes	SB	6.13

Examples are given in Section 6.14.

Each instruction consists of the following information.

- A heading, consisting of the instruction name and mnemonic name
- The op-code
- The syntax definition
- An example of the instruction
- The definition of the instruction
- The status bits affected
- The execution results
- Application notes when appropriate

The op-code is a four-digit hexadecimal number which corresponds to an instruction word whose address fields contain zeros.

The syntax definition follows the conventions described in Section 5. The generic names used in the syntax definitions are:

gas	General Address of the Source operand
gad	General Address of the Destination operand
wa	Workspace register Address
iop	Immediate OPerand
wad	Workspace register Address Destination
disp	DISPlacement of CRU lines from the CRU base register
exp	EXPression that represents an instruction location
cnt	CouNT of bits for CRU transfer
scnt	Shift CouNT
xop	number of eXtended OPeration

Source statements that contain machine instructions can use the label field, the operation field, the operand field, and the comment field.

Use of the label field is optional. When it is used, the label is assigned the address of the instruction. The Assembler advances to the location of a word boundary (even address) before assembling a machine instruction.

The operation (op-code) field contains the mnemonic operation code of the instruction. The contents of the operand field are defined for each instruction.

Inclusion of the comment field is optional. If used, it may contain any ASCII characters, including blanks. The comment has no effect on the assembly process other than being printed in the listing.

In the execution results, the following conventions are used.

( )	Indicates "the contents of."
=>	Indicates "replaces."
* *	Indicates "the absolute value of."

The generic names used in the syntax definitions are also used in the execution results.

6.1 ADD WORDS--A

Op-code: A000 (Format I)

Syntax definition:

[<label>] b A b <gas>,<gad> b [<comment>]

Example:

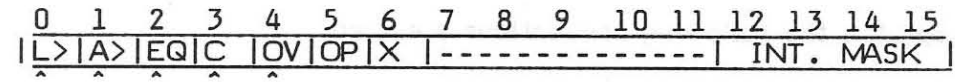
LABEL	A	@ADR1(R2),@ADR2(R3)	Adds the word at the address found by adding ADR1 to the contents of Workspace Register 2 to the word at the address found by adding ADR2 to the contents of Workspace Register 3 and puts the result in the word at the second address.

Definition:

Adds a copy of the source operand (word) to the destination operand (word) and replaces the destination operand with the sum. The computer compares the sum to zero and sets/resets the status bits to indicate the result of the comparison. When there is a carry of bit zero, the carry status bit is set. When there is an overflow, the overflow status bit is set.

Status bits affected:

Logical greater than, arithmetic greater than, equal, carry, and overflow.



Execution results:

(gas) + (gad) => (gad)

Application notes:

The A instruction adds both signed and unsigned integer words. For example, if the address labeled TABLE contains >3124 and Workspace Register 5 contains >8, the instruction

```
A       5,@TABLE
```

results in the contents of TABLE changing to >312C and the contents of Workspace Register 5 not changing. The logical and arithmetic greater than status bits are set and the equal, carry, and overflow status bits are reset.

6.2 ADD BYTES--AB

Op-code: B000 (Format I)

Syntax definition:

[<label>] b AB b <gas>,<gad> b [<comment>]

Example:

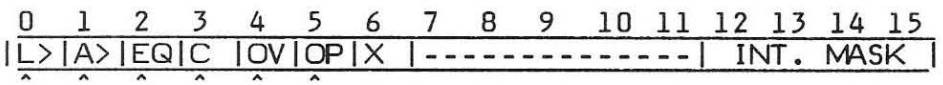
LABEL	AB	3,2	Adds the left byte of Workspace Register 3 to the left byte in Workspace Register 2 and places the result in the left byte of Workspace Register 2.

Definition:

Adds a copy of the source operand (byte) to the destination operand (byte) and replaces the destination operand with the sum. When the source or destination operand is addressed in the Workspace Register mode, only the leftmost byte (bits 0 through 7) of the addressed Workspace Register is used. The computer compares the sum to zero and sets/resets the status bits to indicate the results of the comparison. When there is a carry of the most significant bit of the byte, the carry status bit is set. When there is an overflow, the overflow status bit is set. The odd parity bit is set when the bits in the sum (destination operand) establish odd parity and is reset when the bits in the sum establish even parity.

Status bits affected:

Logical greater than, arithmetic greater than, equal, carry, overflow, and odd parity.



Execution results:

(gas) + (gad) => (gad)

Application notes:

The AB instruction is used to add signed or unsigned integer bytes. For example, if Workspace Register 3 contains >7400, memory word >2122 contains >F318 and Workspace Register 2 contains >2123, the instruction

```
AB      3,*2+
```

changes the contents of memory word >2122 to >F38C because >74 (the value in Workspace Register 3) plus >23 (the value in memory byte >2123) is >8C. The left byte of memory word >2122 is unchanged. The contents of Workspace Register 2 are changed to >2124, while the contents of Workspace Register 3 remain unchanged. The logical greater than, overflow, and odd parity status bits are set, while the arithmetic greater than, equal, and carry status bits are reset.

6.3 ABSOLUTE VALUE--ABS

Op-code: 0740 (Format IV)

Syntax definition:

[<label>] b ABS b <gas> b [<comment>]

Example:

```
LABEL   ABS     *2          Replaces the contents of the word starting at
                            the address in Workspace Register 2 with its
                            absolute value.
```

Definition:

Computes the absolute value of the source operand and replaces the source operand with the result. The absolute value is the two's complement of the source operand when the sign bit (bit zero) is equal to one. When the sign bit is equal to zero, the source operand is unchanged. The computer compares the original source operand to zero and sets/resets the status bits to indicate the results of the comparison.

Status bits affected:

Logical greater than, arithmetic greater than, equal, and overflow.

```
 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
|L>|A>|EQ|C |OV|OP|X |--------------| INT. MASK |
 ^  ^  ^     ^
```

Execution results:

(gas) => (gas)

Application notes:

The ABS instruction is useful for taking the absolute value of an operand. For example, if the third word in array LIST contains the value >FF3C and Workspace Register 7 contains the value >4, the instruction

```
        ABS     @LIST(7)
```

changes the contents of the third word in array LIST to >00C4. The logical greater than status bit is set, while the arithmetic greater than and equal status bits are reset.

6.4 ADD IMMEDIATE--AI

Op-code: 0220 (Format III)

Syntax definition:

[<label>] b AI b <wa>,<iop> b [<comment>]

Example:

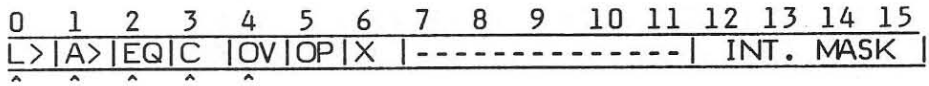
```
LABEL    AI      2,7     Adds 7 to the contents of Workspace Register 2.
```

Definition:

Adds a copy of the immediate operand (the contents of the word following the instruction word in memory) to the contents of the Workspace Register specified in the wa field and replaces the contents of the Workspace Register with the results. The computer compares the sum to zero and sets/resets the status bits to indicate the result of the comparison. When there is a carry of bit zero, the carry status bit is set. When there is an overflow, the overflow status bit is set.

Status bits affected:

Logical greater than, arithmetic greater than, equal, carry, and overflow.



Execution results:

(wa) + iop => (wa)

Application notes:

The AI instruction adds an immediate value to the contents of a Workspace Register. For example, if Workspace Register 6 contains a zero, the instruction

```
AI      6,>C
```

changes the contents of Workspace Register 6 to >000C. The logical greater than and arithmetic greater than status bits are set, while the equal, carry, and overflow status bits are reset.

6.5 DECREMENT--DEC

Op-code: 0600 (Format IV)

Syntax definition:

[<label>] b DEC b <gas> b [<comments>]

Example:

```
LABEL   DEC   2      Decrements the contents of Workspace Register
                     2 by 1.
```

Definition:

Subtracts a value of one from the source operand and replaces the source operand with the result. The computer compares the result to zero and sets/resets the status bits to indicate the result of the comparison. When there is a carry of bit zero, the carry status bit is set. When there is an overflow, the overflow status bit is set.

Status bits affected:

Logical greater than, arithmetic greater than, equal, carry, and overflow.

```
 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
|L>|A>|EQ|C |OV|OP|X |--------------| INT. MASK |
 ^  ^  ^  ^  ^
```

Execution results:

(gas) - 1 => (gas)

Application notes:

The DEC instruction subtracts a value of one from any addressable operand. The DEC instruction is also useful in counting and indexing byte arrays. For example, if COUNT contains a value of >1, the instruction

```
DEC   @COUNT
```

results in a value of zero in location COUNT and sets the equal and carry status bits while resetting the logical greater than, arithmetic greater than, and overflow status bits. The carry bit is always set except on transition from zero to minus one.

6.6 DECREMENT BY TWO--DECT

Op-code: 0640 (Format IV)

Syntax definitions:

[<label>] b DECT b <gas> b [<comment>]

Example:

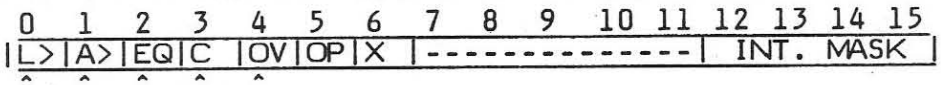
```
LABEL   DECT   @ADDR     Decrements the contents of ADDR by 2.
```

Definition:

Subtracts two from the source operand and replaces the source operand with the result. The computer compares the result to zero and sets/resets the status bits to indicate the result of the comparison. When there is a carry of bit zero, the carry status bit is set. When there is an overflow, the overflow status bit is set.

Status bits affected:

Logical greater than, arithmetic greater than, equal, carry, and overflow.



Execution results:

(gas) - 2 => (gas)

Application notes:

The DECT instruction is useful in counting and indexing word arrays. Also, the DECT instruction enables you to subtract a value of two from any addressable operand. For example, if Workspace Register PRT, which has been equated to 3, contains a value of >2C10, the instruction

```
DECT   PRT
```

changes the contents of Workspace Register 3 to >2C0E. The logical greater than, arithmetic greater than and carry status bits are set, while the equal and overflow status bits are reset.

6.7 DIVIDE--DIV

Op-code: 3C00 (Format IX)

Syntax definition:

[<label>] b DIV b <gas>,<wad> b [<comment>]

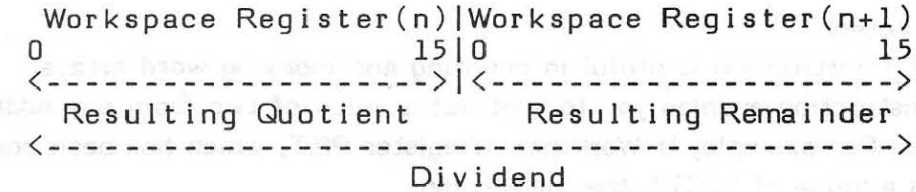
Example:

```
LABEL   DIV     @ADR(2),3   Divides the contents of the words in Workspace
                            Register 3 and Workspace Register 4 by the
                            value of ADR plus Workspace Register 2 and
                            stores the integer result in Workspace Register 3
                            with the remainder in Workspace Register 4.
```

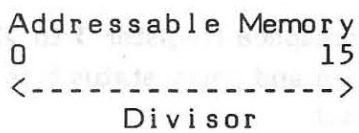
Definition:

Divides the destination operand (a consecutive two-word area of workspace) by a copy of the source operand (one word), using unsigned integer rules. Places the integer quotient in the first of the two-word destination operand area and places the remainder in the second word of that same area. This division is graphically represented as follows.

Destination Operand Workspace Registers:



Source operand:



The first of the destination operand Workspace Registers, shown above, is addressed by the contents of the D field. The dividend is right justified in this

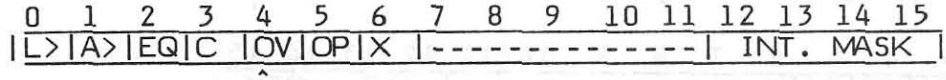
2-word area. When the division is complete, the quotient (result) is placed in the first Workspace Register of the destination operand (represented by n) and the remainder is placed in the second word of the destination operand (represented by n+1).

When the source operand is greater than the first word of the destination operand, normal division occurs. If the source operand is less than or equal to the first word of the destination operand, normal division results in a quotient that cannot be represented in a 16-bit word. In this case, the computer sets the overflow status bit, leaves the destination operand unchanged, and cancels the division operation.

If the destination operand is specified as Workspace Register 15, the first word of the destination operand is Workspace Register 15 and the second word of the destination operand is the word in memory immediately following the workspace area.

Status bits affected:

Overflow



Execution results:

(wad and wad +1) divided by (gas) => (wad) and (wad) + 1

The quotient is placed in wad and the remainder is placed in wad + 1.

Application notes:

The DIV instruction performs a division. For example, if Workspace Register 2 contains a zero and Workspace Register 3 contains >000C, and the contents of LOC is >0005, the instruction

```
DIV     @LOC,2
```

results in >0002 in Workspace Register 2 and >0002 in Workspace Register 3. The overflow status bit is reset. If Workspace Register 2 contained the value >0005, the value contained in the destination operand equals 327,692 and division by the value 5 results in a quotient of 65,538, which cannot be represented in a 16-bit word. This attempted division sets the overflow status bit and the computer cancels the operation.

6.8 INCREMENT--INC

Op-code: 0580 (Format VI)

Syntax definition:

[<label>] b INC b <gas> b [<comment>]

Example:

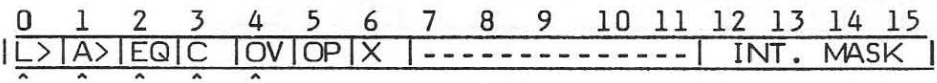
```
LABEL    INC     >1A03        Increments the contents of address >1A03 by 1.
```

Definition:

Adds one to the source operand and replaces the source operand with the result. The computer compares the sum to zero and sets/resets the status bits to indicate the result of the comparison. When there is a carry of bit zero, the carry status bit is set. When there is an overflow, the overflow status bit is set.

Status bits affected:

Logical greater than, arithmetic greater than, equal, carry, and overflow.



Execution results:

(gas) + 1 => (gas)

Application notes:

The INC instruction may be used to count and index byte arrays, add a value of one to an addressable memory location, or set flags. For example, if COUNT contains a zero, the instruction

```
        INC     @COUNT
```

places a >0001 in COUNT and sets the logical greater than and arithmetic greater than status bits, while the equal, carry, and overflow status bits are reset.

6.9 INCREMENT BY TWO--INCT

Op-code: 05C0 (Format VI)

Syntax definition:

[<label>] b INCT b <gas> b [<comment>]

Example:

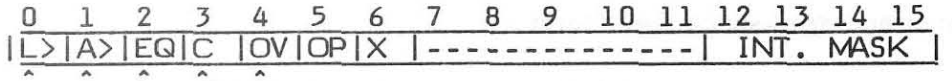
```
LABEL   INCT   3      Increments the contents of Workspace Register
                      3 by 2.
```

Definition:

Adds a value of two to the source operand and replaces the source operand with the sum. The computer compares the sum to zero and sets/resets the status bit to indicate the result of the comparison. When there is a carry of bit zero, the carry status bit is set. When there is an overflow, the overflow status bit is set.

Status bits affected:

Logical greater than, arithmetic greater than, equal, carry, and overflow.



Execution results:

(gas) + 2 => (gas)

Application notes:

The INCT instruction may be used to count and index word arrays and add the value of two to an addressable memory location. For example, if Workspace Register 5 contains the address (>2100) of the fifteenth word of an array, the instruction

```
INCT   5
```

changes Workspace Register 5 to >2102, which points to the sixteenth word of the array. The logical greater than and arithmetic greater than status bits are set, while the equal, carry, and overflow status bits are reset.

6.10 MULTIPLY--MPY

Op-code: 3800 (Format IX)

Syntax definition:

[<label>] b MPY b <gas>,<wad> b [<comment>]

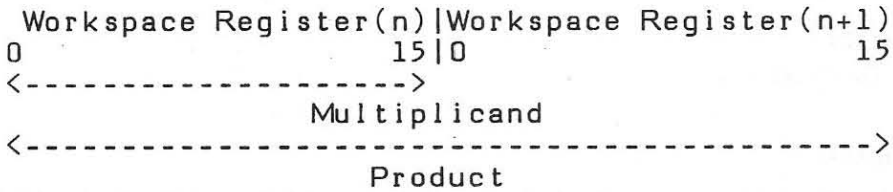
Example:

LABEL	MPY	@ADDR,3	Multiplies the contents of Workspace Register 3 by the value of ADDR. The result is right justified in the 32 bits of Workspace Register 3 and Workspace Register 4.

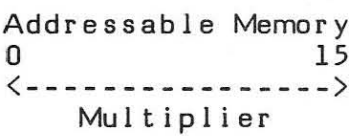
Definition:

Multiplies the first word in the destination operand (a consecutive 2-word area in workspace) by a copy of the source operand and replaces the 2-word destination operand with the result. The multiplication operation may be graphically represented as follows.

Destination operand Workspace Registers:



Source operand:

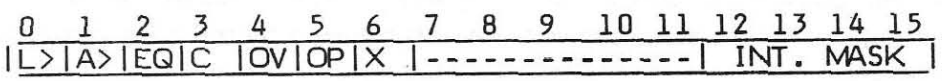


The first word of the destination operand, shown on the previous page, is addressed by the contents of the D field. This word contains the multiplicand (unsigned value of 16 bits) right-justified in the Workspace Register (represented by workspace n above). The 16-bit, unsigned multiplier is located in the source operand. When the multiply operation is complete, the product appears right-justified in the entire 2-word area addressed by the destination field as a 32-bit unsigned value. The maximum value of either input operand is >FFFF and the maximum value of the unsigned product is >FFFE0001.

If the destination operand is specified as Workspace Register 15, the first word of the destination operand is Workspace Register 15 and the second word of the destination operand is the memory word immediately following the workspace memory area.

Status bits affected:

None



Execution results:

(gas) * (wad) = (wad) and (wad)+1

The product (32-bit magnitude) is placed in wad and wad + 1, with the most significant half in wad.

Application notes:

The MPY instruction performs a multiplication. For example, if Workspace Register 5 contains >0012, Workspace Register 6 contains >1B31, and memory location NEW contains >0005, the instruction

```
MPY     @NEW,5
```

changes the contents of Workspace Register 5 to >0000 and Workspace Register 6 to >005A. The source operand is unchanged. The Status Register is not affected by this instruction.

6.11 NEGATE--NEG

Op-code: 0500 (Format VI)

Syntax definition:

[<label>] b NEG b <gas> b [<comment>]

Example:

```
LABEL   NEG   2      Replaces the contents of Workspace Register 2
                     with its additive inverse.
```

Definition:

Replaces the source operand with the two's-complement of the source operand. The computer determines the two's-complement value by inverting all bits of the source operand and adding one to the resulting word. The computer then compares the result to zero and sets/resets the status bits to indicate the result of the comparison.

Status bits affected:

Logical greater than, arithmetic greater than, equal, and overflow.

```
 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
|L>|A>|EQ|C |OV|OP|X |--------------| INT. MASK |
 ^  ^  ^     ^
```

Execution results:

-(gas) => (gas)

Application notes:

The NEG instruction changes the contents of an addressable memory location its additive inverse. For example, if Workspace Register 5 contains the value >A342, the instruction

```
NEG   5
```

changes the contents of Workspace Register 5 to >5CBE. The logical greater than and arithmetic greater than status bits are set, while the equal status bit is reset.

6.12 SUBTRACT WORDS--S

Op-code: 6000 (Format I)

Syntax definition:

[<label>] b S b <gas>,<gad> b [<comment>]

Example:

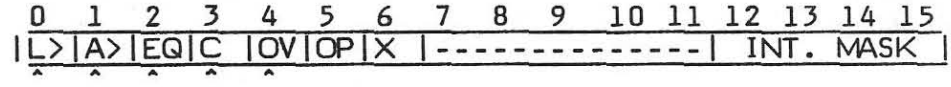
```
LABEL    S       2,3     Subtracts the contents of Workspace Register 2
                         from the contents of Workspace Register 3.
```

Definition:

Subtracts a copy of the source operand from the destination operand and places the difference in the destination operand. The computer compares the difference to zero and sets/resets the status bits to indicate the result of the comparison. When there is a carry of bit zero, the carry status bit is set. When there is an overflow, the overflow status bit is set. The source operand remains unchanged.

Status bits affected:

Logical greater than, arithmetic greater than, equal, carry, and overflow.



Execution results:

(gad) - (gas) => (gad)

Application notes:

The S instruction subtracts signed integer values. For example, if memory location OLDVAL contains a value of >1225 and memory location NEWVAL contains a value of >8223, the instruction

```
        S       @OLDVAL,@NEWVAL
```

changes the contents of NEWVAL to >6FFE. The logical greater than, arithmetic greater than, carry, and overflow status bits are set, while the equal status bit is reset.

6.13 SUBTRACT BYTES--SB

Op-code: 7000 (Format I)

Syntax definition:

[<label>] b SB b <gas>,<gad> b [<comment>]

Example:

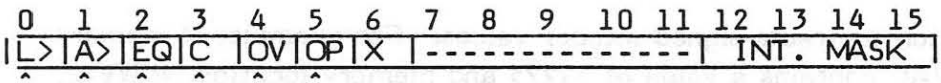
```
LABEL    SB     2,3     Subtracts the leftmost byte of Workspace
                        Register 2 from the leftmost byte of Workspace
                        Register 3.
```

Definition:

Subtracts a copy of the source operand (byte) from the destination operand (byte) and replaces the destination operand byte with the difference. When the destination operand byte is addressed in the Workspace Register mode, only the leftmost byte (bits 0-7) in the Workspace Register is used. The computer compares the resulting byte to zero and sets/resets the status bits accordingly. When there is a carry of the most significant bit of the byte, the carry status bit is set. When there is an overflow, the overflow status bit is set. If the result byte establishes odd parity (an odd number of logic one bits in the byte), the odd parity status bit is set.

Status bits affected:

Logical greater than, arithmetic greater than, equal, carry, overflow, and odd parity.



Execution results:

<gad> - <gas> => <gad>

Application notes:

The SB instruction subtracts signed integer bytes. For example, if Workspace Register 6 contains the value >121C, memory location >121C contains the value >2331, and Workspace Register 1 contains the value >1344, the instruction

```
SB      *6+,1
```

changes the contents of Workspace Register 6 to >121D and the contents of Workspace Register 1 to >F044. The logical greater than status bit is set, while the other status bits affected by this instruction are reset.

6.14 INSTRUCTION EXAMPLES

This section includes several arithmetic instruction examples for further clarification. The application of these instructions is not necessarily limited to that given.

6.14.1 Incrementing and Decrementing Examples

There are two decrement and two increment instructions that may be used for various types of control when passing through a loop, indexing through an array, or operating within a group of instructions.

The incrementing and decrementing instructions available for use with the Assembler are:

INCrement (INC)
INCrement by Two (INCT)
DECrement (DEC)
DECrement by Two (DECT)

The single increment and decrement instructions are useful for indexing byte arrays and for counting byte operations. The increment by two and decrement by two instructions are useful for indexing word arrays and for counting word operations. The following sections provide some examples of these operations.

6.14.1.1 Increment Instruction Example

The example program shows how the INC instruction is useful in byte operations. The program searches a character array for a character with odd parity. To terminate the search, the last character contains zero. The search begins at the lowest address of the array and maintains an index in a Workspace Register. The character array for this example is called A1 and is also the relocatable address of the array. The code is shown on the next page.

```
        SETO    1           Set counter index to -1
SEARCH  INC     1           Increment index
        MOVB    @A1(1),2    Get character
        JOP     ODDP        Jump if found
        JNE     SEARCH      Continue search if not zero
        .
        .
        .
ODDP    ...
```

6.14.1.2 Decrement Instruction Example

To illustrate the use of a DEC instruction in a byte array, this example inverts a 26-character byte array and places the results in another array of the same size called A2. The contents of A1 are defined with a data TEXT statement as follows.

```
A1      TEXT    'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
```

Array A2 is defined with the BSS statement as follows.

```
A2      BSS     26
```

The sample code for the solution is:

```
        LI      5,26            Counter and index for A1.
        LI      4,A2            Address of A2.
INVRT   MOVB    @A1-1(5),*4+    Invert array (Note 1).
        DEC     5               Reduce counter.
        JGT     INVRT           Continue if not complete.
        .
        .
        .
```

Note:

[1]@A1(5) addresses the elements of array A1 in descending order as Workspace Register 5 is decremented. *4+ addresses array A2 in ascending order as Workspace Register four is incremented.

Array A2 contains the following as a result of executing this sequence of code:

```
A2      ZYXWVUTSRQPONMLKJIHGFEDCBA
```

Even though the result of this code sequence is trivial, the use of the MOVB instruction, with indexing by Workspace Register 5 and the result incrementally placed into A2 with the auto-increment function, can be useful in other applications.

The JGT instruction used to terminate the loop allows Workspace Register 5 to serve both as a counter and as an index register.

A special quality of the DEC instruction allows you to simulate a jump greater than or equal to zero instruction. Since DEC always sets the carry status bit except when changing from zero to minus one, it can be used in conjunction with a JOC instruction to form a JGE loop. The example below performs the same function as the preceding example.

```
A1      TEXT    'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
A2      BSS     26
        LI      5,25        Counter and index for A1.
        LI      4,A2        Address of A2.
INVRT   MOVB    @A1(5),*4+  Invert array.
        DEC     5           Reduce counter.
        JOC     INVRT       Continue if not complete.
        .
        .
        .
```

Note: Since the use of JOC makes the loop execute when the counter is zero, the counter is initialized to 25 rather than 26 as in the preceding example.

6.14.1.3 Decrement by Two Instruction Example

To illustrate the use of a DECT instruction in processing word arrays, this example adds the elements of a word array to the elements of another word array and places the results in the second array. The contents of the two arrays are initialized as follows.

```
A1      DATA    500,300,800,1000,1200,498,650,3,27,0
A2      DATA    36,192,517,29,315,807,290,40,130,1320
```

The sample code that adds the two arrays is as follows.

```
        LI      4,20                Initialize counter (Note 1).
SUMS    A       @A1-2(4),@A2-2(4)   Add arrays (Note 2).
        DECT    4                   Decrement counter by two.
        JGT     SUMS                Repeat addition.
```

Notes:

[1]The counter is preset to 20 which is the number of bytes in the array.

[2]The addressing of the two arrays through the use of the at sign (@) is indexed by the counter, which is decremented after each addition.

The contents of the A2 array after the addition process are as follows.

```
A2      536,492,1317,1029,1515,1305,940,43,157,1320
```

There is another method by which this addition process may be accomplished. This method is shown in the following code.

```
        LI      4,10        Initialize counter (Note 1).
        LI      5,A1        Load address of A1 (Note 2).
        LI      6,A2        Load address of A2 (Note 2).
SUMS    A       *5+,*6+     Add arrays (Note 3).
        DEC     4           Decrement counter.
        JGT     SUMS        Repeat addition (Note 4).
```

Notes:

[1]The counter is preset to 10 (the number of elements in the array).

[2]This address is incremented each time an addition takes place. The increment is via the auto-increment function (+).

[3]The * indicates that the contents of the register are to be used as an address, and the + indicates that it is to be automatically incremented by two each time the instruction is executed.

[4]Workspace Register 4 is only greater than zero for ten executions of the DEC instruction, so control is transferred to SUMS nine times after the initial execution.

After execution, the contents of array A2 are the same for this method as for the first.

6.14.2 General Example

The following program illustrates several of the arithmetic instructions. The program consists of a calling program and a subroutine. The subroutine produces the result of the function X-(|3*Y|+5) where X and Y are variable data, treated as signed integers, and passed to the subroutine from the calling program.

To simplify the example, no error checking is included in the subroutine, and it is assumed that the product of 3*Y is in the range of a signed 16-bit word (-32,768 through 32,767).

```
*CALLING PROGRAM
            .
            .
            .
            BL      @CALC       Call subroutine.
VAR         DATA    37          X value.
            DATA    1804        Y value.
            MOV     0,RESULT    Save result.
            .
            .
            .
RESULT      BSS     2
            .
            .
            .
CALC        MOV     *11+,0      Put X value in Register 0.
            MOV     *11,1       Put Y value in Register 1.
            ABS     1           Take absolute value of Y.
            MPY     @THREE,1    Take 3 times absolute value of Y.
            AI      2,5         Add 5 to previous result.
            S       2,0         Subtract previous result from X.
            RT                  Return.
THREE       DATA    3           Constant.
```

SECTION 7: JUMP AND BRANCH INSTRUCTIONS

The following jump and branch instructions are described in this section.

Instruction	Mnemonic	Section
Branch	B	7.1
Branch and Link	BL	7.2
Branch and Load Workspace Pointer	BLWP	7.3
Jump if EQual	JEQ	7.4
Jump if Greater Than	JGT	7.5
Jump if High or Equal	JHE	7.6
Jump if logical High	JH	7.7
Jump if logical Low	JL	7.8
Jump if Low or Equal	JLE	7.9
Jump if Less Than	JLT	7.10
Unconditional JuMP	JMP	7.11
Jump if No Carry	JNC	7.12
Jump if Not Equal	JNE	7.13
Jump if No Overflow	JNO	7.14
Jump if Odd Parity	JOP	7.15
Jump On Carry	JOC	7.16
ReTurn Workspace Pointer	RTWP	7.17
EXecute	X	7.18
EXtended OPeration	XOP	7.19

Examples are given in Section 7.20.

Branch instructions transfer control either unconditionally or conditionally according to the state of one or more bits of the Status Register. The conditional branch (jump) instructions and the status bit or bits tested are shown on the next page.

Status Bits Tested by Jump Instructions

Mnemonic	L>	A>	EQ	C	OV	OP	Jump if:
JH	X	-	X	-	-	-	L>=1 and EQ=0
JL	X	-	X	-	-	-	L>=0 and EQ=0
JHE	X	-	X	-	-	-	L>=1 or EQ=1
JLE	X	-	X	-	-	-	L>=0 or EQ=1
JGT+	-	X	-	-	-	-	A>=1
JLT+	-	X	X	-	-	-	A>=0 and EQ=0
JEQ	-	-	X	-	-	-	EQ=1
JNE	-	-	X	-	-	-	EQ=0
JOC	-	-	-	X	-	-	C=1
JNC	-	-	-	X	-	-	C=0
JNO	-	-	-	-	X	-	OV=0
JOP	-	-	-	-	-	X	OP=1

+Only JGT and JLT use signed arithmetic comparisons. The others are unsigned (logical) comparisons.

For all jump instructions, a displacement of zero results in execution of the next instruction in sequence. A displacement of -1 results in execution of the same instruction (a single-instruction loop).

Each instruction consists of the following information.

- A heading, consisting of the instruction name and mnemonic name
- The op-code
- The syntax definition
- An example of the instruction
- The definition of the instruction
- The status bits affected
- The execution results
- Application notes when appropriate

The op-code is a four-digit hexadecimal number which corresponds to an instruction word whose address fields contain zeros.

The syntax definition follows the conventions described in Section 5. The generic names used in the syntax definitions are:

gas	General Address of the Source operand
gad	General Address of the Destination operand
wa	Workspace register Address
iop	Immediate OPerand
wad	Workspace register Address Destination
disp	DISPlacement of CRU lines from the CRU base register
exp	EXPression that represents an instruction location
cnt	CouNT of bits for CRU transfer
scnt	Shift CouNT
xop	number of eXtended OPeration

Source statements that contain machine instructions can use the label field, the operation field, the operand field, and the comment field.

Use of the label field is optional. When it is used, the label is assigned the address of the instruction. The Assembler advances to the location of a word boundary (even address) before assembling a machine instruction.

The operation (op-code) field contains the mnemonic operation code of the instruction. The contents of the operand field are defined for each instruction.

Inclusion of the comment field is optional. If used, it may contain any ASCII characters, including blanks. The comment has no effect on the assembly process other than being printed in the listing.

In the execution results, the following conventions are used.

( )	Indicates "the contents of."
=>	Indicates "replaces."
* *	Indicates "the absolute value of."

The generic names used in the syntax definitions are also used in the execution results.

7.1 BRANCH--B

Op-code: 0440 (Format VI)

Syntax definition:

[<label>] b B b <gas> b [<comment>]

Example:

LABEL B @THERE Transfers control to location THERE.

Definition:

Replaces the Program Counter contents with the source address and transfers control to the instruction at that location.

Status bits affected:

None.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
L>	A>	EQ	C	OV	OP	X	----------					INT. MASK			

Execution results:

(gas) => (PC)

Application notes:

The B instruction transfers control to another section of code to change the linear flow of the program. For example, if the contents of Workspace Register 3 is >21CC, the instruction

```
B       *3
```

causes the word at location >21CC to be used as the next instruction, because this value replaces the contents of the Program Counter when this instruction is executed.

See Section 24.11.3 for using the B instruction to return to the calling program.

7.2 BRANCH AND LINK--BL

Op-code: 0680 (Format VI)

Syntax definition:

[<label>] b BL b <gas> b [<comment>]

Example:

```
LABEL   BL      @SUBR       Calls SUBR as a common Workspace subroutine.
```

Definition:

Places the source address in the Program Counter, places the address of the instruction following the BL instruction (in memory) in Workspace Register 11, and transfers control to the new Program Counter contents.

Status bits affected:

None.

```
 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
|L>|A>|EQ|C |OV|OP|X |--------------| INT. MASK |
```

Execution results:

(old PC) => (Workspace Register 11)
(gas) => (PC)

Application notes:

The BL instruction returns linkage. For example, if the instruction

```
        BL      @TRAN
```

occurs at memory location >04BC, this instruction has the effect of placing memory location TRAN in the Program Counter. Since the instruction BL @TRAN requires two words of machine code (which are placed at addresses >04BC and 04BE), the word address immediately following the second word is >04C0 so that value is the address placed in Workspace Register 11.

7.3 BRANCH AND LOAD WORKSPACE POINTER--BLWP

Op-code: 0400 (Format VI)

Syntax definition:

[<label>] b BLWP b <gas> b [<comment>]

Example:

```
LABEL   BLWP   @VECT      Branches to subroutine at address (@VECT+2)
                          and executes context switch.
```

Definition:

Places the source operand in the Workspace Pointer and the word immediately following the source operand in the Program Counter. Places the previous contents of the Workspace Pointer in the new Workspace Register 13, places the previous contents of the Program Counter (address of the instruction following BLWP) in the new Workspace Register 14, and places the contents of the Status Register in the new Workspace Register 15. When all store operations are complete, the computer transfers control to the new Program Counter.

Status bits affected:

None.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
L>	A>	EQ	C	OV	OP	X	----------					INT. MASK			

Execution results:

(gas) => (WP)
(gas + 2) => (PC)
(old WP) => (Workspace Register 13)
(old PC) => (Workspace Register 14)
(ST) => (Workspace Register 15)

Application notes:

The BLWP instruction links to subroutines, program modules, or other programs that do not necessarily share the calling program's workspace. See Section 7.20.3 for an example of using the BLWP instruction.

7.4 JUMP IF EQUAL--JEQ

Op-code: 1300 (Format II)

Syntax definition:

[<label>] b JEQ b <exp> b [<comment>]

Example:

```
LABEL   JEQ     LOC         Jumps to LOC if EQ = 1.
```

Definition:

When the equal status bit is set, transfers control by adding the signed displacement in the instruction word to the Program Counter and then placing the sum in the Program Counter to transfer control.

Status bits tested:

Equal.

```
 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
|L>|A>|EQ|C |OV|OP|X |--------------| INT. MASK |
```

Jump if: EQ = 1

Status bits affected:

None.

```
 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
|L>|A>|EQ|C |OV|OP|X |--------------| INT. MASK |
```

Execution results:

If the equal bit is equal to 1: (PC) + Displacement => (PC).
If the equal bit is equal to 0: (PC) => (PC).

Application notes:

The JEQ instruction transfers control when the equal status bit is set.

7.5 JUMP IF GREATER THAN--JGT

Op-code: 1500 (Format II)

Syntax definition:

[<label>] b JGT b <exp> b [<comment>]

Example:

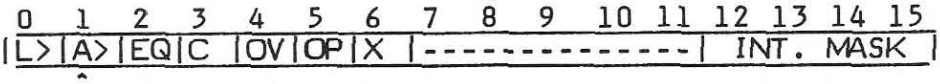
```
LABEL    JGT     THERE      Jumps to THERE if A> = 1.
```

Definition:

When the arithmetic greater than status bit is set, adds the signed displacement in the instruction word to the Program Counter and places the sum in the Program Counter. Transfers control to the new Program Counter location.

Status bits tested:

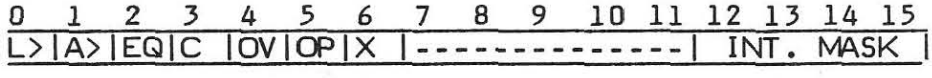
Arithmetic greater than.



Jump if: A> = 1

Status bit affected:

None.



Execution results:

If the arithmetic greater than bit is equal to 1: (PC) + Displacement => (PC).
If the arithmetic greater than bit is equal to 0: (PC) => (PC).

Application notes:

The JGT instruction transfers control if the arithmetic greater than status bit is set.

7.6 JUMP IF HIGH OR EQUAL--JHE

Op-code: 1400 (Format II)

Syntax definition:

[<label>] b JHE b <exp> b [<comment>]

Example:

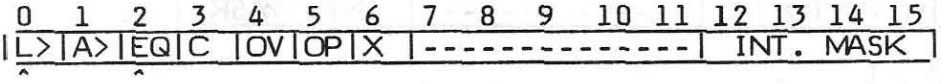
LABEL	JHE	BLBD	Jumps to location BLBD if either EQ or L> is set.

Definition:

When the equal status bit or the logical greater than status bit is set, adds the signed displacement in the instruction word to the Program Counter and replaces the contents of the Program Counter with the sum.

Status bits tested:

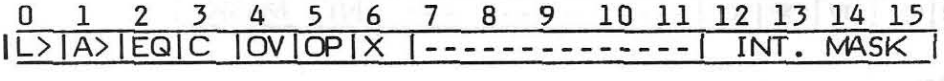
Logical greater than, equal.



Jump if: L> = 1 or EQ = 1

Status bits affected:

None.



Execution results:

If the logical greater than bit is equal to 1 or the equal bit is equal to 1: (PC) + Displacement => (PC).

If the logical greater than bit and the equal bit are equal to 0: (PC) => (PC).

Application notes:

The JHE instruction transfers control when either the logical greater than or equal status bit is set.

7.7 JUMP IF LOGICAL HIGH--JH

Op-code: 1B00 (Format II)

Syntax definition:

[<label>] b JH b <exp> b [<comment>]

Example:

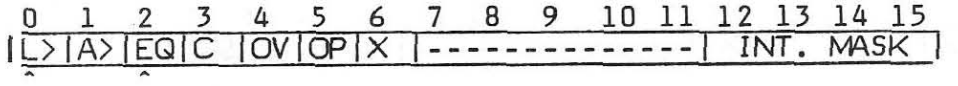
LABEL JH CONT If L> equals 1 and EQ equals 0, skips to CONT.

Definition:

When the equal status bit is reset and the logical greater than status bit is set, adds the signed displacement in the instruction word to the contents of the Program Counter and replaces the Program Counter with the sum.

Status bits tested:

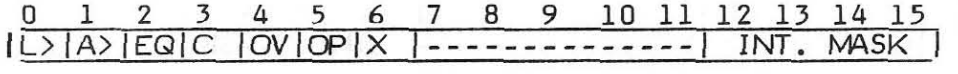
Logical greater than, equal.



Jump if: L> = 1 and EQ = 0

Status bits affected:

None.



Execution results:

If the logical greater than bit is equal to 1 and the equal bit is equal to 0:
(PC) + Displacement => (PC).
If the logical greater than bit is equal to 0 or the equal bit is equal to 1:
(PC) => (PC).

Application notes:

The JH instruction transfers control when the equal status bit is reset and the logical greater than status bit is set.

7.8 JUMP IF LOGICAL LOW--JL

Op-code: 1A00 (Format II)

Syntax definition:

[<label>] b JL b <exp> b [<comment>]

Example:

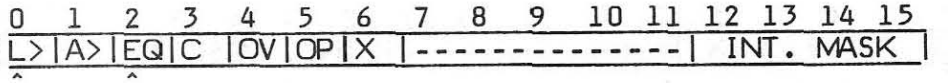
LABEL JL PREVLB If L> and EQ are reset, jumps to PREVLB.

Definition:

When the equal and logical greater than status bits are reset, adds the signed displacement in the instruction word to the Program Counter contents and replaces the Program Counter with the sum.

Status bits tested:

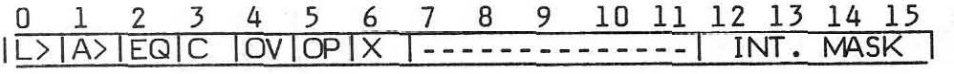
Logical greater than, equal.



Jump if: L> = 0 and EQ = 0

Status bits affected:

None.



Execution results:

If the logical greater than bit and the equal bit are equal to 0:
(PC) + Displacement => (PC).
If the logical greater than bit is equal to 1 or the equal bit is equal to 1:
(PC) => (PC).

Application notes:

The JL instruction transfers control when the equal and logical greater than status bits are reset.

7.9 JUMP IF LOW OR EQUAL--JLE

Op-code: 1200 (Format II)

Syntax definition:

[<label>] b JLE b <exp> b [<comment>]

Example:

```
LABEL    JLE      THERE      Jumps to THERE when EQ = 1 or L> = 0.
```

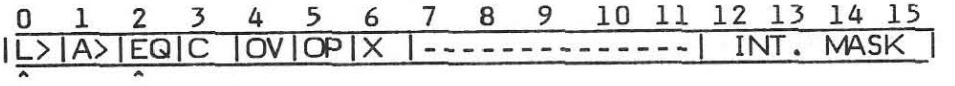
Definition:

When the equal status bit is set or the logical greater than status bit is reset, adds the signed displacement in the instruction word to the contents of the Program Counter and replaces the Program Counter with the sum.

Note: JLE is not "jump if less than or equal."

Status bits tested:

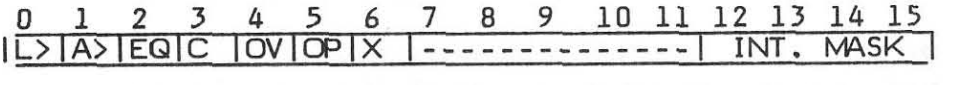
Logical greater than, equal.



Jump if: L> = 0 or EQ = 1

Status bits affected:

None.



Execution results:

If the logical greater than bit is equal to 0 or the equal bit is equal to 1:
(PC) + Displacement => (PC).
If the logical greater than bit is equal to 1 and the equal bit is equal to 0:
(PC) => (PC).

Application notes:

The JLE instruction transfers control when the equal status bit is set or the logical greater than status bit is reset.

7.10 JUMP IF LESS THAN--JLT

Op-code: 1100 (Format II)

Syntax definition:

[<label>] b JLT b <exp> b [<comment>]

Example:

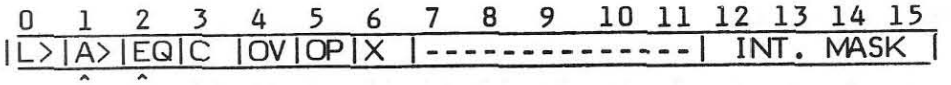
LABEL JLT THERE Jumps to THERE if A> = 0 and EQ = 0.

Definition:

When the equal and arithmetic greater than status bits are reset, adds the signed displacement in the instruction word to the Program Counter and replaces the Program Counter contents with the sum.

Status bits tested:

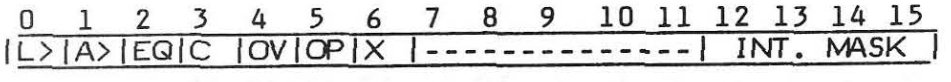
Arithmetic greater than, equal.



Jump if: A> = 0 and EQ = 0

Status bits affected:

None.



Execution results:

If the arithmetic greater than bit and the equal bit are equal to 0:
(PC) + *Displacement* => (PC)
If the arithmetic greater than bit is equal to 1 or the equal bit is equal to 1:
(PC) => (PC).

Application notes:

The JLT instruction transfers control when the equal and arithmetic greater than status bits are reset.

7.11 UNCONDITIONAL JUMP--JMP

Op-code: 1000 (Format II)

Syntax definition:

[<label>] b JMP b <exp> b [<comment>]

Example:

LEAVE	JMP	>11A3	Jumps to address >11A3 if it is within >100 bytes of the current address.

Definition:

Adds the signed displacement in the instruction word to the Program Counter and replaces the Program Counter with the sum if the sum is within >100 bytes of the current Program Counter.

Status bits affected:

None.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
L>	A>	EQ	C	OV	OP	X	-	-	-	-	-	INT. MASK			

Execution results:

(PC) + Displacement => (PC)

The Program Counter is always incremented to the address of the next instruction prior to execution of an instruction. The execution results of jump instructions refer to the Program Counter contents after the contents have been incremented to address the next instruction in sequence. The displacement (in words) is shifted to the left one bit position to orient the word displacement to the word address, and added to the Program Counter contents. The sum must be within >100 bytes of the current Program Counter.

Application notes:

The JMP instruction transfers control to another section of the program.

7.12 JUMP IF NO CARRY--JNC

Op-code: 1700 (Format II)

Syntax definition:

[<label>] b JNC b <exp> b [<comment>]

Example:

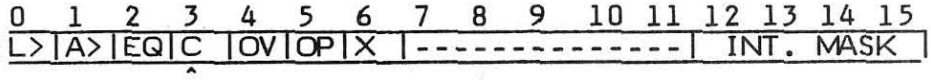
LABEL JNC NONE Jumps to NONE if C = 0.

Definition:

When the carry status bit is reset, adds the signed displacement in the instruction word to the Program Counter and replaces the Program Counter with the sum.

Status bits tested:

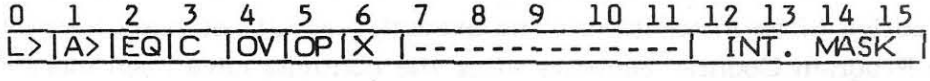
Carry.



Jump if: C = 0

Status bits affected:

None.



Execution results:

If the carry bit is equal to 0: (PC) + Displacement => (PC).

If the carry bit is equal to 1: (PC) => (PC).

Application notes:

The JNC instruction transfers control when the carry status bit is reset.

7.13 JUMP IF NOT EQUAL--JNE

Op-code: 1600 (Format II)

Syntax definition:

[<label>] b JNE b <exp> b [<comment>]

Example:

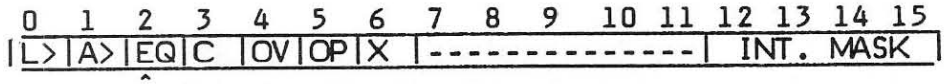
```
LABEL    JNE     LOC2        Jumps to LOC2 if EQ = 0.
```

Definition:

When the equal status bit is reset, adds the signed displacement in the instruction word to the Program Counter and replaces the Program Counter with the sum.

Status bits tested:

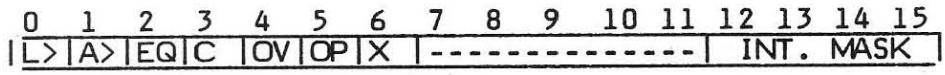
Equal.



Jump if: EQ = 0

Status bits affected:

None.



Execution results:

If the equal bit is equal to 0: (PC) + Displacement => (PC).
If the equal bit is equal to 1: (PC) => (PC).

Application notes:

The JNE instruction transfers control when the equal status bit is reset. For instance, JNE is often useful when testing CRU bits.

7.14 JUMP IF NO OVERFLOW--JNO

Op-code: 1900 (Format II)

Syntax definition:

[<label>] b JNO b <exp> b [<comment>]

Example:

```
LABEL    JNO     NORML     Jumps to NORML if OV = 0.
```

Definition:

When the overflow status bit is reset, adds the displacement in the instruction word to the Program Counter and replaces the Program Counter with the sum.

Status bits tested:

Overflow.

```
 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
|L>|A>|EQ|C |OV|OP|X |--------------| INT. MASK |
          ^
```

Jump if: OV = 0

Status bits affected:

None.

```
 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
|L>|A>|EQ|C |OV|OP|X |--------------| INT. MASK |
```

Execution results:

If the overflow bit is equal to 0: (PC) + Displacement => (PC).
If the overflow bit is equal to 1: (PC) => (PC).

Application notes:

The JNO instruction transfers control when the overflow status bit is reset. JNO normally transfers control during arithmetic sequences where addition, subtraction, incrementing, and decrementing may cause an overflow condition. JNO may also be used following an SLA (Shift Left Arithmetic) operation. If, during SLA execution, the sign of the Workspace Register being shifted changes, the overflow status bit is set. This feature permits transfer, after a sign change, to error correction routines or to another functional code sequence.

7.15 JUMP IF ODD PARITY--JOP

Op-code: 1C00 (Format II)

Syntax definition:

[<label>] b JOP b <exp> b [<comment>]

Example:

LABEL JNP THERE Jumps to THERE if OP = 1.

Definition:

When the odd parity status bit is set, adds the signed displacement in the instruction word to the Program Counter and replaces the Program Counter with the sum.

Status bits tested:

Odd parity.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
L>	A>	EQ	C	OV	OP	X	----------					INT. MASK			

Jump if: OP = 1

Status bits affected:

None.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
L>	A>	EQ	C	OV	OP	X	----------					INT. MASK			

Execution results:

If the odd parity bit is equal to 1: (PC) + Displacement => (PC).
If the odd parity bit is equal to 0: (PC) => (PC).

Application notes:

The JOP instruction transfers control when there is odd parity. Odd parity indicates that there is an odd number of logic one bits in the byte tested. JOP transfers control if the byte tested contains an odd number of logic one bits. This instruction may be used in data transmissions where the parity of the transmitted byte is used to ensure the validity of the received character at the point of reception.

7.16 JUMP ON CARRY--JOC

Op-code: 1800 (Format II)

Syntax definition:

[<label>] b JOC b <exp> b [<comment>]

Example:

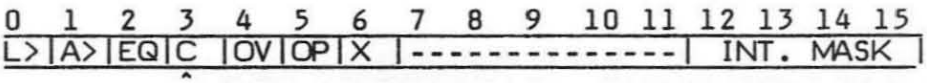
LABEL JOC PROCED If C = 1, jumps to PROCED.

Definition:

When the carry status bit is set, adds the signed displacement in the instruction word to the Program Counter and replaces the Program Counter with the sum.

Status bits tested:

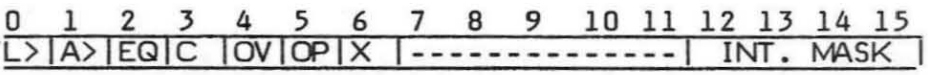
Carry.



Jump if: C = 1

Status bits affected:

None.



Execution results:

If the carry bit is equal to 1: (PC) + Displacement => (PC).
If the carry bit is equal to 0: (PC) => (PC).

Application notes:

The JOC instruction transfers control when the carry status bit is set.

7.17 RETURN WITH WORKSPACE POINTER--RTWP

Op-code: 0380 (Format VII)

Syntax definition:

[<label>] b RTWP b [<comment>]

Example:

```
LABEL   RTWP   Returns from subroutine called by BLWP.
```

Definition:

Replaces the contents of the Workspace Pointer Register with the contents of the current Workspace Register 13. Replaces the contents of the Program Counter with the contents of the current Workspace Register 14. Replaces the contents of the Status Register with the contents of the current Workspace Register 15. The effect of this instruction is to restore the execution environment that existed prior to an interrupt, a BLWP instruction, or an XOP instruction.

Status bits affected:

Restores all status bits to the value contained in Workspace Register 15.

```
 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
|L>|A>|EQ|C |OV|OP|X |--------------| INT. MASK |
 ^  ^  ^  ^  ^  ^  ^                 ^  ^  ^  ^
```

Execution results:

(Workspace Register 13) => (WP)
(Workspace Register 14) => (PC)
(Workspace Register 15) => (ST)

Application notes:

The RTWP instruction restores the execution environment after the completion of an interrupt, a BLWP instruction, or an XOP instruction.

7.18 EXECUTE--X

Op-code: 0480 (Format VI)

Syntax definition:

[<label>] b X b <gas> b [<comment>]

Example:

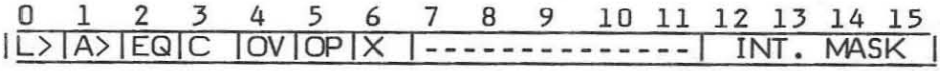
LABEL X 2 Executes the contents of Workspace Register 2.

Definition:

Executes the source operand as an instruction. When the source operand is not a single word instruction, the word or words following the execute instruction are used with the source operand as a 2-word or 3-word instruction. The source operand, when executed as an instruction, may affect the contents of the Status Register. The Program Counter increments by either one, two, or three words depending upon the source operand. If the executed instruction is a branch, the branch is taken. If the executed instruction is a jump and if the conditions for a jump (i.e. the status test indicates a jump) are satisfied, then the jump is taken relative to the location of the X instruction.

Status bits affected:

None, but substituted instruction affects status bits normally.



Execution results:

An instruction at gas is executed instead of the X instruction.

Application notes:

The X instruction executes the source operand as an instruction. This is primarily useful when the instruction to be executed is dependent upon a variable factor. Refer to Section 7.20 for additional application notes.

7.19 EXTENDED OPERATION--XOP

Op-code: 2C00 (Format IX)

Syntax definition:

[<label>] b XOP b <gas>,<xop> b [<comment>]

Example:

LABEL	XOP	@BUFF(4),1	Performs XOP 1 on the word of the address BUFF plus the displacement specified by Workspace Register 4.

Definition:

This instruction is on all TI-99/4A Home Computers. However, some only support XOP 2 while others support both XOP 1 and XOP 2. To find out if your TI-99/4A computer supports the XOP 1 instruction, run CALL PEEK in TI BASIC and read one word at address >44. If the word if >FFD8, then XOP 1 is available. If it contains other data (most likely >FFE8), then XOP 1 is not available.

The op field specifies the extended operation transfer vector in memory. The two memory words at that location contain the Workspace Pointer and Program Counter contents for the software implemented XOP instruction subroutine. Note that the two memory words at this location must contain the necessary Workspace Pointer and Program Counter values prior to the XOP instruction execution for software implemented instructions.

XOP 1 is at address >44, with vectors >FFD8 and >FFF8. XOP 2 is at address >48 with vectors >83A0 and >8300. The first entry in the vector is the new workspace address. The second entry is the new Program Counter address.

When the computer is turned on, XOP 1 is set up to be used with development software used by Texas Instruments. However, if you have XOP 1 you may modify the data for your own use.

The effective address of the source operand is placed in Workspace Register 11 of the XOP workspace. The Workspace Pointer contents are placed in Workspace Register 13 of the XOP workspace. The Program Counter contents are placed in Workspace Register 14 of the XOP workspace. The Status contents are placed in

Workspace Register 15 of the XOP workspace. *Control is transferred to the new* Program Counter address and the software implemented XOP is executed. (XOP execution of software implemented XOP instruction is similar to an interrupt trap execution.)

Status bits affected:

Extended operation.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
L>	A>	EQ	C	OV	OP	X	----------					INT. MASK			

Execution results:

gas => (Workspace Register 11)
(>0040 + (op)*4) => (WP)
(>0042 + (op)*4) => (PC)
(WP) => (Workspace Register 13)
(PC) => (Workspace Register 14)
(ST) => (Workspace Register 15)
1 => X (XOP status bit)

7.20 INSTRUCTION EXAMPLES

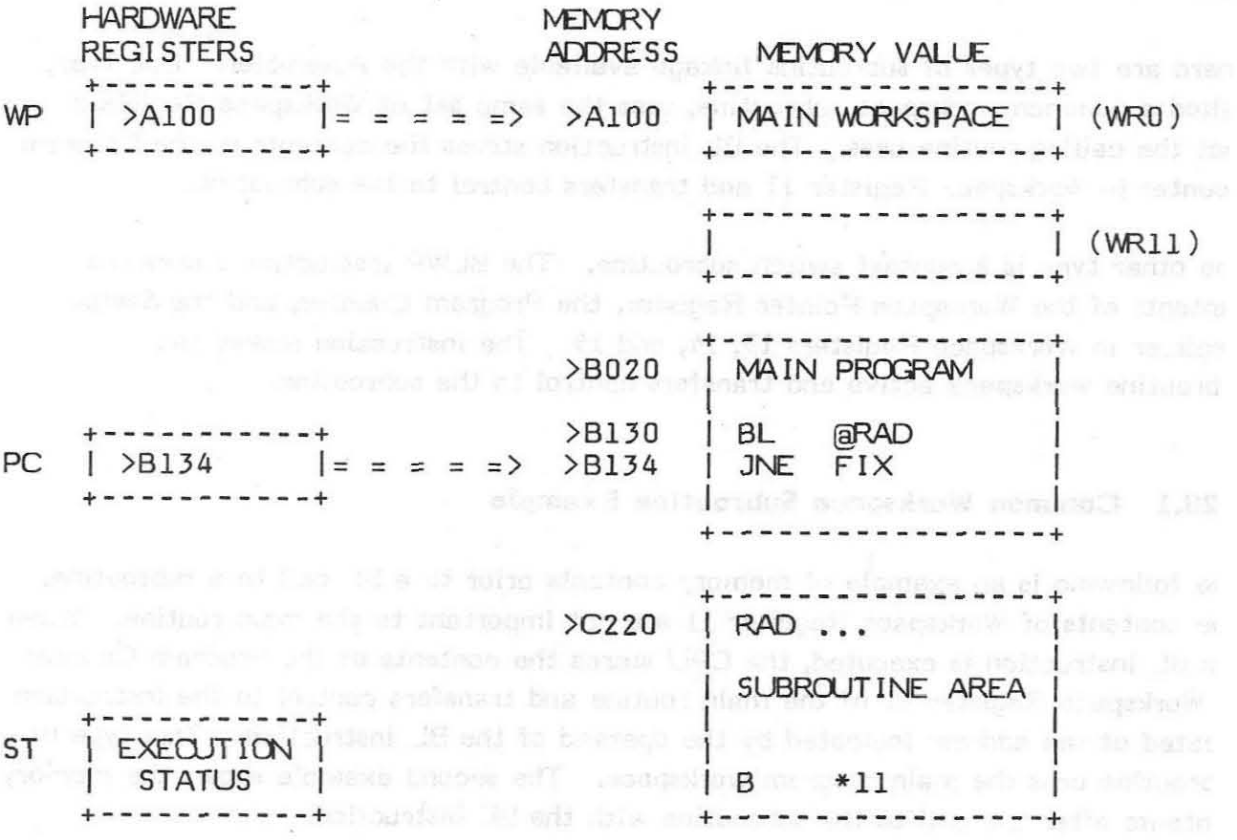
There are two types of subroutine linkage available with the Assembler. One type, called a common workspace subroutine, uses the same set of Workspace Registers that the calling routine uses. The BL instruction stores the contents of the Program Counter in Workspace Register 11 and transfers control to the subroutine.

The other type is a context switch subroutine. The BLWP instruction stores the contents of the Workspace Pointer Register, the Program Counter, and the Status Register in Workspace Registers 13, 14, and 15. The instruction makes the subroutine workspace active and transfers control to the subroutine.

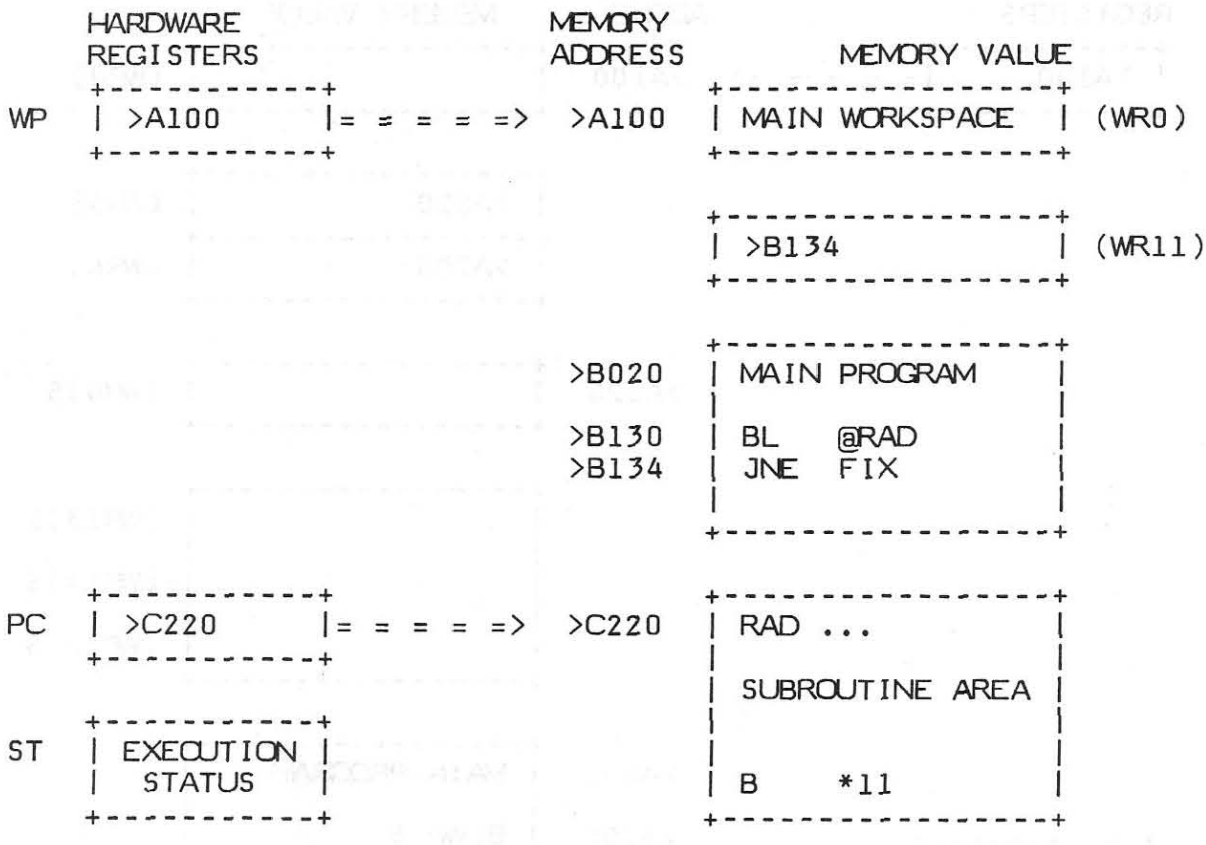
7.20.1 Common Workspace Subroutine Example

The following is an example of memory contents prior to a BL call to a subroutine. The contents of Workspace Register 11 are not important to the main routine. When the BL instruction is executed, the CPU stores the contents of the Program Counter in Workspace Register 11 of the main routine and transfers control to the instruction located at the address indicated by the operand of the BL instruction. This type of subroutine uses the main program workspace. The second example shows the memory contents after the call to the subroutine with the BL instruction.

When the instruction at location >1130 is executed (BL @RAD), the present contents of the Program Counter, which point to the next instruction, are saved in Workspace Register 11. Workspace Register 11 would then contain an address of >1134. The Program Counter is then loaded with the address of label RAD, which is address >2220. This subroutine returns to the main program with a branch to the address in Workspace Register 11 using the B *11 instruction.



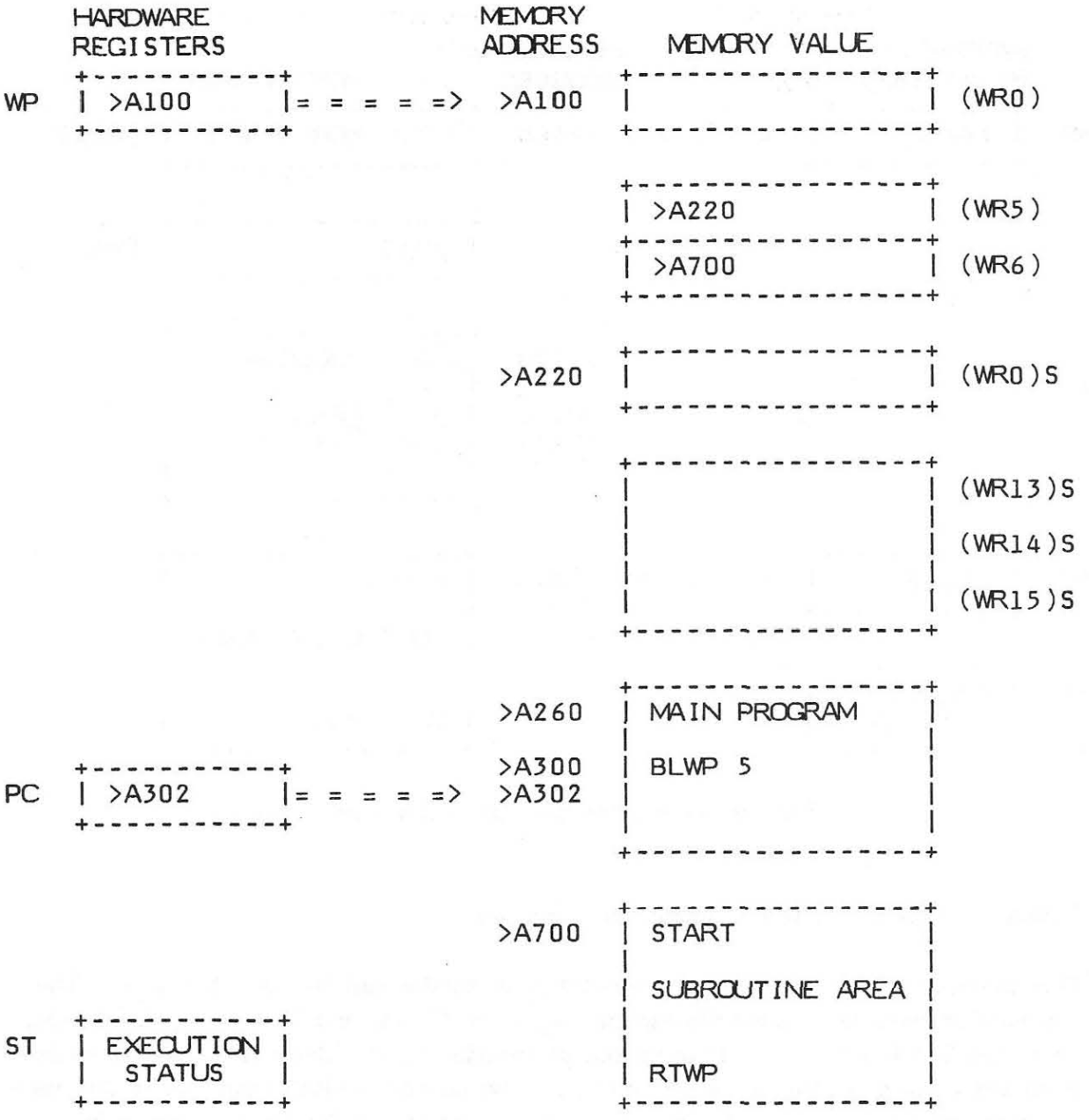
Common Workspace Subroutine Example



PC Contents after BL Instruction Execution

7.20.2 Context Switch Subroutine Example

This example shows the memory contents prior to the call to the subroutine. The contents of the subroutine's Workspace Registers 13, 14, and 15 are not significant. When the BLWP instruction is executed at location >0300, there is a context switch from the main program to the subroutine. The context switch then places the main program Program Counter, Workspace Pointer, and Status Register contents in Workspace Registers 13, 14, and 15 of the subroutine. This saves the environment of the main program for use on return. The operand of the BLWP instruction specifies that the address vector for the context switch is in Workspace Registers 5 and 6. The address in Workspace Register 5 is placed in the Workspace Pointer Register, and the address in Workspace Register 6 is placed in the Program Counter.

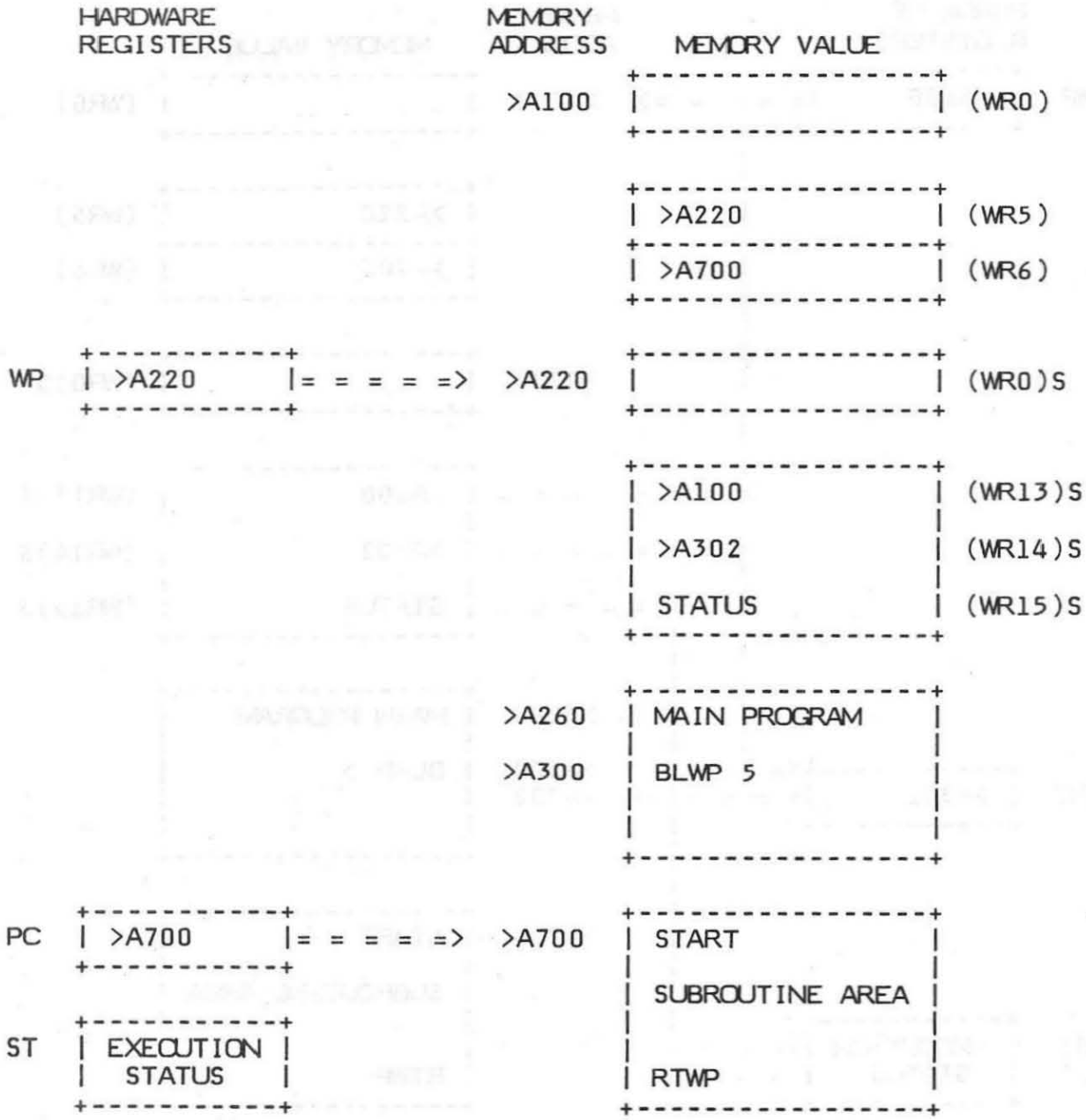


(WNr) = Workspace Register of Main Program
(WNr)S = Workspace Register of Subroutine

Context Switch Subroutine Example

After the instruction at location >0300 is executed, the Workspace Pointer points to the subroutine workspace and the Program Counter points to the first instruction of the subroutine. The contents of the Status Register are not reset prior to the

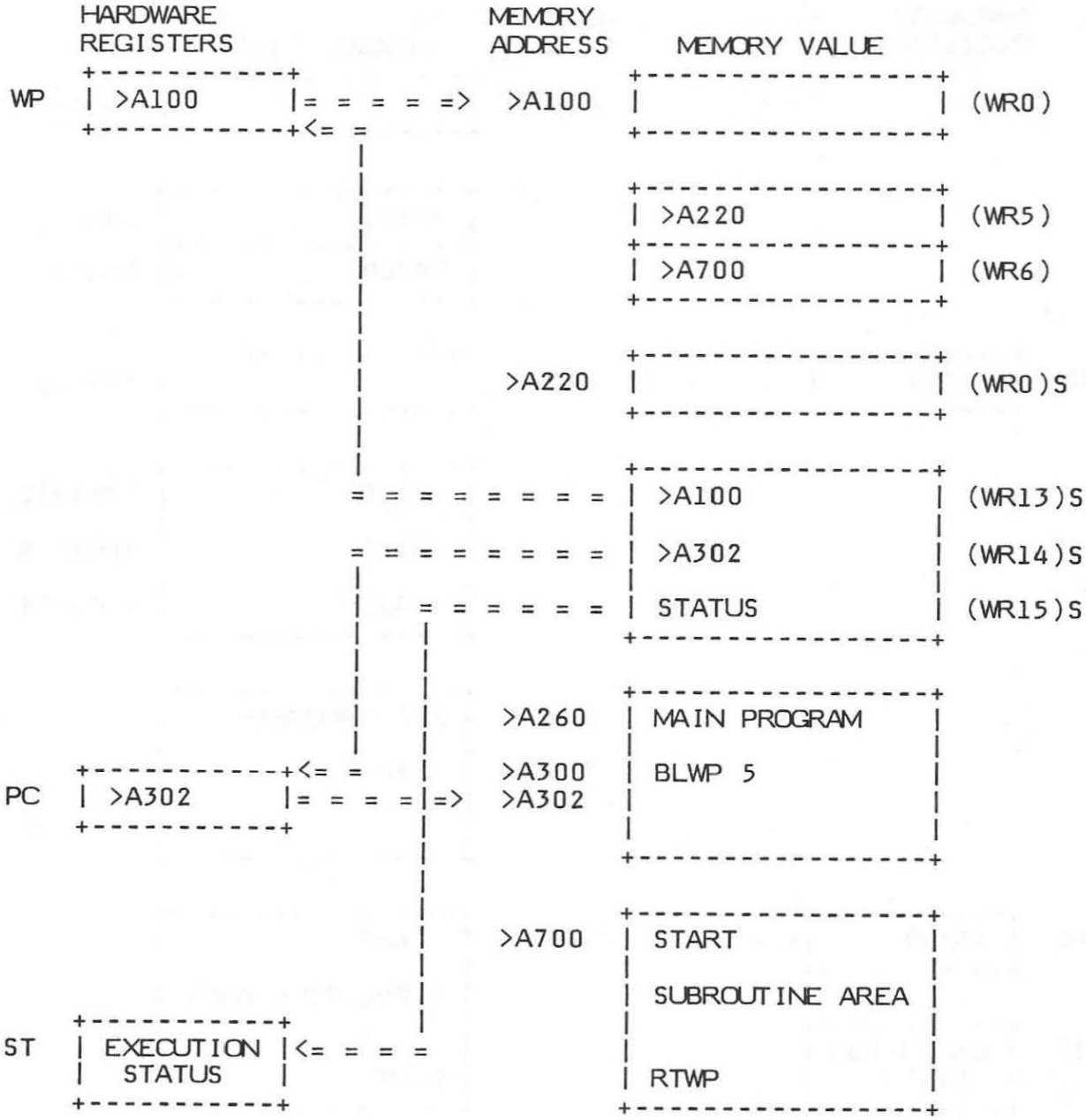
execution of the first instruction of the subroutine, so the status indicated will actually be the status of the main program execution. A subroutine may then execute depending on the status of the main program.



(WNr) = Workspace Register of Main Program
(WNr)S = Workspace Register of Subroutine

After Execution of BLWP Instruction

This example subroutine contains a *RTWP return from the subroutine. Control is* transferred to the main program at the instruction following the BLWP to the subroutine. The Status Register is restored from Workspace Register 15 and the Workspace Pointer points to the workspace of the main program.



(WNr) = Workspace Register of Main Program
(WNr)S = Workspace Register of Subroutine

After Return using the RTWP Instruction

7.20.3 Passing Data to Subroutines

When a subroutine is entered with a context switch (BLWP), data may be passed using either the contents of Workspace Register 13 or 14 of the subroutine workspace. Workspace Register 13 contains the memory address of the calling program's workspace, which may contain data to be passed to the subroutine. Workspace Register 14 contains the memory address of the next memory location following the BLWP instruction. This location and following locations may also contain data to be passed to the subroutine.

When the calling program's workspace contains data for the subroutine, this data may be obtained by using the indexed memory address mode indexed by Workspace Register 13. The address used is equal to twice the number of the Workspace Register that contains the desired data. The following instruction is an example.

```
MOV     @10(13),R10
```

The contents of Workspace Register 5 of the calling program's workspace (bytes 10 and 11 relative to the workspace address) are placed in Workspace Register 10 of the subroutine workspace.

The following examples show the passing of data to a subroutine by placing the data following the BLWP instruction.

```
        BLWP    @SUB        Subroutine call.
        DATA    V1          Data.
        DATA    V2          Data.
        DATA    V3          Data.
        JEQ     ERROR       Return from subroutine, test for
        .                   error.  (The subroutine sets the
        .                   equal status bit to one for error.)
        .
SUB     DATA    SUBWS,SUBPRG  Entry point for SUB & SUB
                              Workspace.
        .
        .
        .
SUBWS   BSS     32
SUBPRG  MOV     *14+,1      Fetch V1 placed in Workspace
                            Register 1.
        MOV     *14+,2      Fetch V2 placed in Workspace
                            Register 2.
        MOV     *14+,3      Fetch V3 placed in Workspace
                            Register 3.
        .
        .
        .
        RTWP                Return from subroutine.
```

The three MOV instructions retrieve the variables from the main program module and place them in Workspace Registers one, two, and three of the subroutine.

When the BLWP instruction is executed, the main program module status is stored in Workspace Register 15 of the subroutine. If the subroutine returns with a RTWP instruction, this status is placed in the Status Register after the RTWP instruction is executed. The subroutine may alter the Status Register contents prior to executing the RTWP instruction. The calling program can then test the appropriate bit of the status word (the equal bit in this example) with jump instructions.

A BL instruction can also be used to pass parameters to a subroutine. When using this instruction, the originating Program Counter value is placed in Workspace Register 11. Therefore, the subroutine must fetch the parameters relative to the contents of Workspace Register 11 rather than the contents of Workspace Register 14 as in the BLWP example. The following example demonstrates parameter passing with a BL instruction.

```
        BL      @SUBR           Branch to subroutine.
        DATA    PARM1,PARM2     Passed parameters stored in next
                                two memory words.
        JEQ     ERROR           Test for error. (Subroutine sets the
                                equal status bit to one for error.)
        .
        .
        .
SUBR    EQU     $
        MOV     *R11+,R0        Get value of first parameter and put
                                in Workspace Register 0.
        MOV     *R11+,R1        Get value of second parameter and
                                put in Workspace Register 1. (R11
                                is incremented past the locations of
                                the two data words and now
                                indicates the address of the next
                                instruction in the main program.)
        .
        .
        .
        B       *11
```

7.20.4 Extended Operations

Extended operation instructions permit a limited extension of the existing instruction set to include additional instructions. In the computer, these additional instructions are implemented by software routines.

When the program module contains an XOP instruction that is software implemented, the computer locates the XOP Workspace Pointer and Program Counter words in the XOP reserved memory locations and loads the Workspace Pointer and Program Counter. When the Workspace Pointer and Program Counter are loaded, the computer transfers control to the XOP instruction set through a context switch. When the context switch is complete, the XOP workspace contains the calling routine return data in Workspace Registers 13, 14, and 15.

The XOP instruction passes one operand to the XOP (input to the XOP routine in Workspace Register 11 of the XOP workspace). At the completion of the software XOP, the XOP routine should return to the calling routine with an RTWP instruction that restores the execution environment of the calling routine to that in existence at the call to the XOP.

7.20.5 Execute Example

The execute instruction may be used to execute an instruction that is not in sequence without transferring control to the desired instruction. One useful application is to execute one of a table of instructions, selecting the desired instruction by using an index into the table. The computed value of the index determines which instruction is executed.

A table of shift instructions illustrates the use of the X instruction. Place the following instructions at location TBLE.

```
TBLE    SLA    R6,3      Shift Workspace Register 6.
        SLA    R7,3      Shift Workspace Register 7.
        SLA    R8,3      Shift Workspace Register 8.
TABEND  EQU    $
```

A character is placed in the most significant byte of Workspace Register 5 to select the Workspace Register to be shifted to the left 3 bit positions. ASCII characters A, B, and C specify shifting Workspace Registers 6, 7, and 8, respectively. Other characters are ignored. The following code performs the selection of the shift desired.

```
        SRL     R5,8                Move to lower byte.
        AI      R5,-'A'             Subtract table bias.
        JLT     NOSHIFT             Illegal.
        SLA     R5,1                Make it a word index.
        CI      R5,TABEND-TBLE-2
        JGT     NOSHFT              Illegal.
        X       @TBLE(R5)
NOSHFT  EQU     $
        .
        .
        .
```

When using the X instruction, if the substituted instruction contains a Ts field or a Td field that results in a two word instruction, the computer accesses the word following the X instruction as the second word, not the word following the substituted instruction. When the substituted instruction is a jump instruction with a displacement, the displacement must be computed from the X instruction, not from the substituted instruction.

SECTION 8: COMPARE INSTRUCTIONS

The following compare instructions are described in this section.

Instruction	Mnemonic	Section
Compare words	C	8.1
Compare Bytes	CB	8.2
Compare Immediate	CI	8.3
Compare Ones Corresponding	COC	8.4
Compare Zeros Corresponding	CZC	8.5

Compare instructions have no effect other than the setting or resetting of appropriate status bits in the Status Register. The compare instructions perform both arithmetic and logical comparisons. An arithmetic comparison is of the two operands as two's complement values, while a logical comparison is of the two operands as unsigned magnitude values.

Each instruction consists of the following information.

- A heading, consisting of the instruction name and mnemonic name
- The op-code
- The syntax definition
- An example of the instruction
- The definition of the instruction
- The status bits affected
- The execution results
- Application notes when appropriate

The op-code is a four-digit hexadecimal number which corresponds to an instruction word whose address fields contain zeros.

The syntax definition follows the conventions described in Section 5. *The generic* names used in the syntax definitions are:

gas	General Address of the Source operand
gad	General Address of the Destination operand
wa	Workspace register Address
iop	Immediate OPerand
wad	Workspace register Address Destination
disp	DISPlacement of CRU lines from the CRU base register
exp	EXPression that represents an instruction location
cnt	CouNT of bits for CRU transfer
scnt	Shift CouNT
xop	number of eXtended OPeration

Source statements that contain machine instructions can use the label field, the operation field, the operand field, and the comment field.

Use of the label field is optional. When it is used, the label is assigned the address of the instruction. The Assembler advances to the location of a word boundary (even address) before assembling a machine instruction.

The operation (op-code) field contains the mnemonic operation code of the instruction. The contents of the operand field are defined for each instruction.

Inclusion of the comment field is optional. If used, it may contain any ASCII characters, including blanks. The comment has no effect on the assembly process other than being printed in the listing.

In the execution results, the following conventions are used.

( )	Indicates "the contents of."
=>	Indicates "replaces."
* *	Indicates "the absolute value of."

The generic names used in the syntax definitions are also used in the execution results.

8.1 COMPARE WORDS--C

Op-code: 8000 (Format I)

Syntax definition:

[<label>] b C b <gas>,<gad> b [<comment>]

Example:

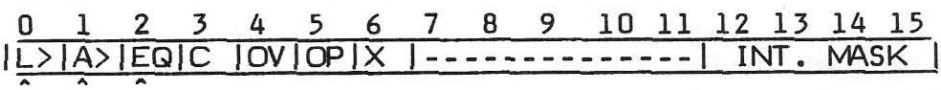
LABEL	C	2,3	Compares the contents of Workspace Register 2 and Workspace Register 3.

Definition:

Compares the source operand (word) with the destination operand (word) and sets/resets the status bits to indicate the results of the comparison. The arithmetic and equal comparisons compare the operand as signed, two's complement values. The logical comparison compares the two operands as unsigned, 16-bit magnitude values.

Status bits affected:

Logical greater than, arithmetic greater than, and equal.



Execution results:

(gas) compared to (gad)

Application notes:

The C instruction compares the two operands as signed, two's complement values and as unsigned integers. Some examples are:

		Status Bits Set		
Source	Destination	Logical>	Arithmetic>	Equal
>FFFF	>0000	1	0	0
>7FFF	>0000	1	1	0
>8000	>0000	1	0	0
>8000	>7FFF	1	0	0
>7FFF	>7FFF	0	0	1
>7FFF	>8000	0	1	0

An alternate way to compare a word or byte to zero is to move the word or byte to itself. For example:

```
MOV     R0,R0
JEQ     OUT
```

jumps to OUT if R0 is equal to zero.

8.2 COMPARE BYTES--CB

Op-code: 9000 (Format I)

Syntax definition:

[<label>] b CB b <gas>,<gad> b [<comment>]

Example:

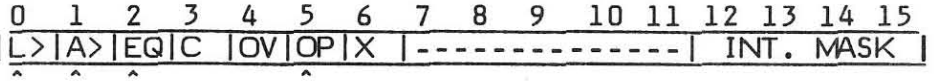
LABEL	CB	2,3	Compares the leftmost bytes of Workspace Register 2 and Workspace Register 3.

Definition:

Compares the source operand (byte) with the destination operand (byte) and sets/resets the status bits according to the result of the comparison. The CB instruction uses the same comparison basis as does the C instruction. If the source operand contains an odd number of logic one bits, the odd parity status bit is set. The operands remain unchanged. If either operand is addressed in the Workspace Register mode, the byte addressed is the most significant byte.

Status bits affected:

Logical greater than, arithmetic greater than, equal, and odd parity.



Execution results:

(gas) compared to (gad)

Application notes:

The CB instruction compares the two operands as signed, two's complement values or as unsigned integers. Some examples are:

		Status Bits Set			
Source	Destination	Logical>	Arithmetic>	Equal	Odd Parity
>00	>FF	1	0	0	0
>00	>7F	1	1	0	1
>7F	>80	1	0	0	1
>7F	>7F	0	0	1	1
>80	>7F	0	1	0	1

8.3 COMPARE IMMEDIATE--CI

Op-code: 0280 (Format VIII)

Syntax definition:

[<label>] b CI b <wa>,<iop> b [<comment>]

Example:

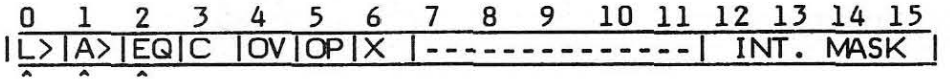
LABEL	CI	3,7	Compares the contents of Workspace Register 3 to >0007.

Definition:

Compares the contents of the specified Workspace Register with the word in memory immediately following the instruction and sets/resets the status bits according to the comparison. The CI instruction makes the same type of comparison as does the C instruction.

Status bits affected:

Logical greater than, arithmetic greater than, and equal.



Execution results:

(wa) compared to iop

Application notes:

The CI instruction compares the Workspace Register to an immediate operand. For example, if the contents of Workspace Register 9 is >2183, the instruction

```
CI      9,>F330
```

results in the arithmetic greater than status bit being set and the logical greater than and equal status bits being reset.

8.4 COMPARE ONES CORRESPONDING--COC

Op-code: 2000 (Format III)

Syntax definition:

[<label>] b COC b <gas>,<wad> b [<comment>]

Example:

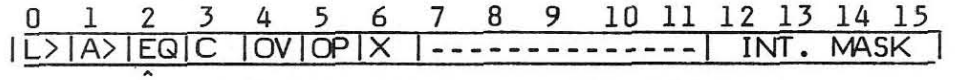
LABEL	COC	@MASK,2	Compares the contents of Workspace Register 2 with the contents of MASK.

Definition:

When the bits in the destination operand Workspace Register that correspond to the logic one bits in the source operand are equal to logic one, sets the equal status bit. The source and destination operands are unchanged.

Status bits affected:

Equal.



Execution results:

The equal bit is set if all bits of <wad> that correspond to the bits of <gas> that are equal to 1 are also equal to 1.

Application notes:

The COC instruction tests single or multiple bits within a word in a Workspace Register. For example, if TESTBI contains the word >C102 and Workspace Register 8 contains the value >E306, the instruction

```
COC     @TESTBI,8
```

sets the equal status bit because for each 1 bit in the first operand there is a 1 bit in the corresponding bit position of the second operand as shown below.

```
>C102 = 1100 0001 0000 0010 and
>E306 = 1110 0011 0000 0110
```

If Workspace Register 8 contains >E301, the equal status bit is reset. Use this instruction to determine if a Workspace Register has 1s in the bit positions indicated by the 1s in a mask.

8.5 COMPARE ZEROS CORRESPONDING--CZC

Op-code: 2400 (Format III)

Syntax definition:

[<label>] b CZC b <gas>,<wad> b [<comment>]

Example:

LABEL	CZC	@MASK,2	Compares the contents of Workspace Register 2 with the contents of MASK.

Definition:

When the bits in the destination operand Workspace Register that correspond to the one bits in the source operand are all equal to logic zero, sets the equal status bit. The source and destination operands are unchanged.

Status bits affected:

Equal.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
L>	A>	EQ	C	OV	OP	X	-	-	-	-	-	INT. MASK			

Execution results:

The equal bit is set if all bits of <wad> that correspond to the bits of <gas> that are equal to 1 are equal to 0.

Application notes:

The CZC instruction tests single or multiple bits within a word in a Workspace Register. For example, if the memory location labeled TESTBI contains the value >C102, and Workspace Register 8 contains >2301, the instruction

```
CZC     @TESTBI,8
```

resets the equal status bit because for each 1 bit in the first operand there is not a corresponding zero bit in the corresponding bit position of the second operand as shown below.

```
>C102 = 1100 0001 0000 0010 and
>2301 = 0010 0011 0000 0001
```

If Workspace Register 8 contains the value >2201, then the equal status bit is set. Use the CZC instruction to determine if a Workspace Register has zeros in the positions indicated by ones in a mask.

SECTION 9: CONTROL AND CRU INSTRUCTIONS

The following control and CRU instructions are described in this section.

Instruction	Mnemonic	Section
LoaD CRU	LDCR	9.1
Set CRU Bit to One	SBO	9.2
Set CRU Bit to Zero	SBZ	9.3
STore CRU	STCR	9.4
Test Bit	TB	9.5

The following instructions are described in Section 9.6. All of them are properly assembled and are recognized by the TMS9900 microprocessor, but they should not be used on the Home Computer.

Instruction	Mnemonic
ClocK OFf	CKOF
ClocK ON	CKON
IDLE	IDLE
ReSET	RSET
Load or REstart eXecution	LREX

Examples are given in Section 9.7.

Control instructions affect the operation of the Arithmetic Unit (AU) and the associated portions of the computer or microprocessor. CRU instructions affect the modules connected to the Communications Register Unit.

For CRU bit instructions, the signed displacement is shifted one bit position to the left and added to the contents of Workspace Register 12. In other words, it is a displacement in bits from the contents of bits 3 through 14 of Workspace Register 12.

See Section 24.3.2 for more information.

Each instruction consists of the following information.

- A heading, consisting of the instruction name and mnemonic name
- The op-code
- The syntax definition
- An example of the instruction
- The definition of the instruction
- The status bits affected
- The execution results
- Application notes when appropriate

The op-code is a four-digit hexadecimal number which corresponds to an instruction word whose address fields contain zeros.

The syntax definition follows the conventions described in Section 5. The generic names used in the syntax definitions are:

gas	General Address of the Source operand
gad	General Address of the Destination operand
wa	Workspace register Address
iop	Immediate OPerand
wad	Workspace register Address Destination
disp	DISPlacement of CRU lines from the CRU base register
exp	EXPression that represents an instruction location
cnt	CouNT of bits for CRU transfer
scnt	Shift CouNT
xop	number of eXtended OPeration

Source statements that contain machine instructions can use the label field, the operation field, the operand field, and the comment field.

Use of the label field is optional. When it is used, the label is assigned the address of the instruction. The Assembler advances to the location of a word boundary (even address) before assembling a machine instruction.

The operation (op-code) field contains the mnemonic operation code of the instruction. The contents of the operand field are defined for each instruction.

Inclusion of the comment field is optional. If used, it may contain any ASCII characters, including blanks. The comment has no effect on the assembly process other than being printed in the listing.

In the execution results, the following conventions are used.

( ) Indicates "the contents of."
=> Indicates "replaces."
* * Indicates "the absolute value of."

The generic names used in the syntax definitions are also used in the execution results.

9.1 LOAD CRU--LDCR

Op-code: 3000 (Format IV)

Syntax definition:

[<label>] b LDCR b <gas>,<cnt> b [<comment>]

Example:

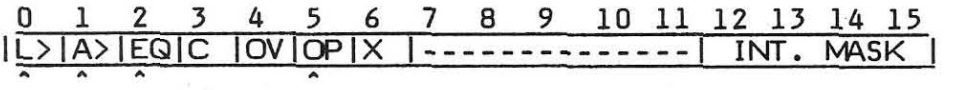
```
WRITE   LDCR   @BUFF,15   Sends 15 bits from BUFF to the CRU.
```

Definition:

Transfers the number of bits specified in the cnt field from the source operand to the CRU. The transfer begins with the least significant bit of the source operand. The CRU address is contained in bits 3 through 14 of Workspace Register 12. When the cnt field contains zero, the number of bits transferred is 16. If the number of bits to be transferred is from one to eight, the source operand address is a byte address. If the number of bits to be transferred is from 9 to 16, the source operand address is a word address. If the source operand address is odd, the address is truncated to an even address prior to data transfer. When the number of bits transferred is a byte or less, the source operand is compared to zero and the status bits are set/reset, according to the results of the comparison. The odd parity status bit is set when the bits in a byte (or less) to be transferred establish odd parity.

Status bits affected:

Logical greater than, arithmetic greater than, and equal. When cnt is less than nine, odd parity is also set or reset. The odd parity status bit is set according to the full word or byte, not just the transferred bits.



Execution results:

The number of bits specified by cnt are transferred from memory at address gas to consecutive CRU lines beginning at the address in Workspace Register 12 (bits 3 through 14).

9.2 SET CRU BIT TO ONE--SBO

Op-code: 1D00 (Format II)

Syntax definition:

[<label>] b SBO b <disp> b [<comment>]

Example:

```
LABEL   SBO     7       Sets CRU bit 7, relative to the CRU base in
                        Workspace Register 12, to one.
```

Definition:

Sets the digital output bit to one on the CRU at the address derived from this instruction. The derived address is the sum of the signed displacement and the contents of Workspace Register 12, bits 3 through 14. The execution of this instruction does not affect the Status Register or the contents of Workspace Register 12.

Status bits affected:

None.

```
 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
|L>|A>|EQ|C |OV|OP|X |--------------| INT. MASK |
```

Execution results:

A CRU bit is set to one. The CRU bit equals the sum of the contents of Workspace Register 12 (bits 3 through 14) and the displacement.

9.3 SET CRU BIT TO ZERO--SBZ

Op-code: 1E00 (Format II)

Syntax definition:

[<label>] b SBZ b <disp> b [<comment>]

Example:

```
LABEL   SBZ     7       Sets CRU bit 7, relative to the CRU base in
                        Workspace Register 12, to zero.
```

Definition:

Sets the digital output bit to zero on the CRU at the address derived from this instruction. The derived address is the sum of the signed displacement and the contents of Workspace Register 12, bits 3 through 14. The execution of this instruction does not affect the Status Register or the contents of Workspace Register 12.

Status bits affected:

None.

```
 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
|L>|A>|EQ|C |OV|OP|X |--------------| INT. MASK |
```

Execution results:

A CRU bit is set to zero. The CRU bit equals the sum of the contents of Workspace Register 12 (bits 3 through 14) and the displacement.

9.4 STORE CRU--STCR

Op-code: 3400 (Format IV)

Syntax definition:

[<label>] b STCR b <gas>,<cnt> b [<comment>]

Example:

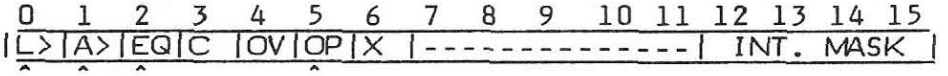
READ	STCR	@BUF,9	Reads 9 bits from the CRU and stores them at location BUF.

Definition:

Transfers the number of bits specified in the cnt field from the CRU to the source operand. The transfer begins from the CRU address specified in bits 3 through 14 of Workspace Register 12 to the least significant bit of the source operand and fills the source operand toward the most significant bit. When the cnt field contains a zero, the number of bits to transfer is 16. If the number of bits to transfer is from one to eight, the source operand address is a byte address. Any bit in the memory byte not filled by the transfer is set to zero. When the number of bits to transfer is from 9 to 16, the source operand address is a word address. If the source operand address is odd, the address is truncated to an even address prior to data transfer. If the transfer does not fill the entire memory word, unfilled bits are set to zero. When the number of bits to transfer is a byte or less, the bits transferred are compared to zero and the status bits are set or reset to indicate the results of the comparison. Also, when the bits to be transferred are a byte or less, the odd parity bit is set when the bits establish odd parity.

Status bits affected:

Logical greater than, arithmetic greater than, and equal. When cnt is less than 9, odd parity is also set or reset. Status is set according to the full word or byte, not just the transferred bits.



Execution results:

The number of bits specified by cnt are transferred from consecutive CRU lines beginning at the address in Workspace Register 12 (bits 3 through 14) to memory at address gas.

Application notes:

The STCR instruction transfers a specified number of CRU bits from the CRU to the memory location specified as the source operand. Note that the CRU base address must be in Workspace Register 12 (bits 3 through 14) prior to the execution of this instruction.

9.5 TEST BIT--TB

Op-code: 1F00 (Format II)

Syntax definition:

[<label>] b TB b <disp> b [<comment>]

Example:

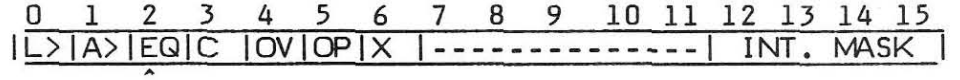
CHECK	TB	7	Reads CRU bit 7 relative to the CRU base address in Workspace Register 12, and sets the equal status bit to the value read.

Definition:

Reads the digital input bit on the CRU at the address specified by the sum of the signed displacement and the contents of Workspace Register 12, bits 3 through 14, and set the equal status bit to the value read. The digital input bit and the contents of Workspace Register 12 are unchanged.

Status bits affected:

Equal.



Execution results:

Equal bit is set to the value of the CRU bit addressed by the sum of the contents of Workspace Register 12 (bits 3 through 14) and the displacement.

Application notes:

The TB instruction transfers the level from the indicated CRU line to the equal status bit without modification. If the CRU line tested is set to one, the equal status bit is set to one; if the line is zero, it is set to zero. The JEQ instruction can then be used to transfer control when the CRU line is one and not transfer control when the line is zero. In addition, the JNE instruction transfers control under the opposite conditions.

9.6 OTHER INSTRUCTIONS

The following instructions are properly assembled and are recognized by the TMS9900 microprocessor, but they should not be used on the Home Computer. Their op-code and syntax definition are given below.

Instruction	Mnemonic	Op-code	Format	Syntax definition
ClocK OFf	CKOF	03C0	VII	[<label>] b CKOF b [<comment>
ClocK ON	CKON	03A0	VII	[<label>] b CKON b [<comment>]
IDLE	IDLE	0340	VII	[<label>] b IDLE b [<comment>]
ReSET	RSET	0360	VII	[<label>] b RSET b [<comment>]
Load or REstart eXecution	LREX	03E0	VII	[<label>] b LREX b [<comment>]

9.7 CRU INPUT/OUTPUT

The Communications Register Unit (CRU) performs single and multiple bit programmed input/output. All input consists of reading CRU line logic levels into memory and output consists of setting CRU output lines to bit values from a word or byte of memory. The CRU provides a maximum of 4096 input and output lines that may be individually selected by a 12-bit address. The 12-bit address is located in bits 3 through 14 of Workspace Register 12 and is the base address for all CRU communications. See Section 24.3.2 for more information.

9.7.1 CRU I/O Instructions

There are five instructions for communications with CRU lines.

SBO Set CRU Bit to One. This instruction sets a CRU output line to one.

SBZ Set CRU Bit to Zero. This instruction sets a CRU output line to zero.

TB Test CRU Bit. This instruction reads the digital input bit and sets the equal status bit (bit 2) to the value of the digital input bit.

LDCR Load Communications Register. This instruction transfers the number of bits (1-16) specified by the cnt field of the instruction to the CRU from the source operand. When less than nine bits are specified, the source operand address is a byte address. When nine or more bits are specified, the source operand is a word address. The CRU address is the address of the first CRU digital output affected. The CRU address is determined by the contents of Workspace Register 12, bits 3 through 14.

STCR Store Communications Register. This instruction transfers the number of bits specified by the cnt field of the instruction from the CRU to the source operand. When less than nine bits are specified, the source operand address is a byte address. When nine or more bits are specified, the source operand address is a word address. The CRU address is determined by Workspace Register 12, bits 3 through 14.

9.7.2 Accessing Specific Bits

There are many different ways to access the same CRU bit. For instance, if Workspace Register 12 contains >0100, making the base address in bits 3 through 14 equal to >80, the following instruction sets CRU line >85 to one.

```
SBO     5
```

Alternatively, if Workspace Register 12 contains >010A, making the base address in bits 3 through 14 equal to >85, the following instruction also sets CRU line >85 to one.

```
SBO     0
```

9.7.3 SBO Example

Assume that a control device turns on a motor when the computer sets a one on CRU line >10F and that Workspace Register 12 contains >0200, making the base address in bits 3 through 14 equal to >100. The following instruction sets CRU line >10F to one.

```
SBO     15
```

9.7.4 SBZ Example

Assume that a control device shuts off a valve when the computer sets a zero on a CRU line is connected to CRU line 2 and that Workspace Register 12 contains zero. The following instruction sets CRU line 2 to zero.

```
SBZ     2
```

9.7.5 TB Example

Assume that Workspace Register 12 contains >0140, making the base address in bits 3 through 14 equal to >A0. The following instructions test the input on CRU line >A4 and execute the instructions beginning at location RUN when the CRU line is set to one. When the CRU line is set to zero, the instructions beginning at location WAIT are executed.

```
        TB      4       Test CRU line 4.
        JEQ     RUN     If on, go to RUN.
WAIT    .               If off, continue.
        .
        .
RUN     .
```

The TB instruction sets the equal bit of the Status Register to the level on line 4 of the CRU device.

SECTION 10: LOAD AND MOVE INSTRUCTIONS

The following load and move instructions are described in this section.

Instruction	Mnemonic	Section
Load Immediate	LI	10.1
Load Interrupt Mask Immediate	LIMI	10.2
Load Workspace Pointer Immediate	LWPI	10.3
MOVe words	MOV	10.4
MOVe Bytes	MOVB	10.5
STore STatus	STST	10.6
STore Workspace Pointer	STWP	10.7
SWaP Bytes	SWPB	10.8

An example is given in Section 10.9.

Load and move instructions permit you to establish the execution environment and the execution results. These instructions manipulate data between memory locations and between hardware registers and memory locations.

Each instruction consists of the following information.

- A heading, consisting of the instruction name and mnemonic name
- The op-code
- The syntax definition
- An example of the instruction
- The definition of the instruction
- The status bits affected
- The execution results
- Application notes when appropriate

The op-code is a four-digit hexadecimal number which corresponds to an instruction word whose address fields contain zeros.

LOAD AND MOVE INSTRUCTIONS

The syntax definition follows the conventions described in Section 5. The generic names used in the syntax definitions are:

gas	General Address of the Source operand
gad	General Address of the Destination operand
wa	Workspace register Address
iop	Immediate OPerand
wad	Workspace register Address Destination
disp	DISPlacement of CRU lines from the CRU base register
exp	EXPression that represents an instruction location
cnt	CouNT of bits for CRU transfer
scnt	Shift CouNT
xop	number of eXtended OPeration

Source statements that contain machine instructions can use the label field, the operation field, the operand field, and the comment field.

Use of the label field is optional. When it is used, the label is assigned the address of the instruction. The Assembler advances to the location of a word boundary (even address) before assembling a machine instruction.

The operation (op-code) field contains the mnemonic operation code of the instruction. The contents of the operand field are defined for each instruction.

Inclusion of the comment field is optional. If used, it may contain any ASCII characters, including blanks. The comment has no effect on the assembly process other than being printed in the listing.

In the execution results, the following conventions are used.

( )	Indicates "the contents of."
=>	Indicates "replaces."
* *	Indicates "the absolute value of."

The generic names used in the syntax definitions are also used in the execution results.

10.1 LOAD IMMEDIATE--LI

Op-code: 0200 (Format VIII)

Syntax definition:

[<label>] b LI b <wa>,<iop> b [<comment>]

Example:

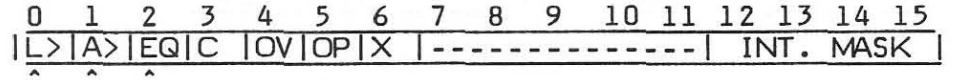
```
GETIT    LI      3,>17        Loads Workspace Register 3 with >0017.
```

Definition:

Places the immediate operand (the word of memory immediately following the instruction) in the Workspace Register (W field). The immediate operand is not affected by the execution of this instruction. The immediate operand is compared to 0 and the logical greater than, arithmetic greater than, and equal status bits are set or reset according to the result of the comparison.

Status bits affected:

Logical greater than, arithmetic greater than, and equal.



Execution results:

(iop) => (wa)

Application notes:

The LI instruction places an immediate operand in a specified Workspace Register. *This may be used to initialize a Workspace Register as a loop counter.* For example, the instruction

```
        LI      7,5
```

initializes Workspace Register 7 with the value >0005. In this example, the logical greater than and arithmetic greater than status bits are set, while the equal status bit is reset.

10.2 LOAD INTERRUPT MASK IMMEDIATE--LIMI

Op-code: 0300 (Format VIII)

Syntax definition:

[<label>] b LIMI b <iop> b [<comment>]

Example:

```
LABEL   LIMI    2           Masks level 2 and below.
```

Definition:

Places the least significant four bits (bits 12-15) of the contents of the immediate operand (the next word after the instruction) in the interrupt mask of the Status Register. The remaining bits of the Status Register (0 through 11) are not affected.

Status bits affected:

Interrupt mask.

```
 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
|L>|A>|EQ|C |OV|OP|X |--------------| INT. MASK |
                                      ^  ^  ^  ^
```

Execution results:

Places the four least significant bits of iop into the interrupt mask.

Application notes:

The LIMI instruction initializes the interrupt mask so that a particular level of interrupt is accepted. For example, the instruction

```
        LIMI    2
```

sets the interrupt mask to level two and enables interrupts at levels 0, 1, and 2.

The instruction

```
        LIMI 0
```

Disables all interrupts and is the normal state of the computer.

10.3 LOAD WORKSPACE POINTER IMMEDIATE--LWPI

Op-code: 02E0 (Format VIII)

Syntax definition:

[<label>] b LWPI b <iop> b [<comment>]

Example:

```
NEWWP   LWPI    >02F2       Sets NEWWP equal to >02F2.
```

Definition:

Replaces the contents of the Workspace Pointer with the immediate operand.

Status bits affected:

None.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
L>	A>	EQ	C	OV	OP	X	----------					INT. MASK			

Execution results:

(iop) => (WP)

Application notes:

The LWPI instruction initializes or changes the Workspace Pointer Register to alter the Workspace environment of the program. You may use a BLWP or a LWPI instruction to load your own Workspace Registers.

10.4 MOVE WORD--MOV

Op-code: C000 (Format I)

Syntax definition:

[<label>] b MOV b <gas>,<gad> b [<comment>]

Example:

```
GET     MOV     @WD,2       Moves a copy of WD into Workspace Register 2.
```

Definition:

Replaces the destination operand with a copy of the source operand. The computer compares the resulting destination operand to zero and sets/resets the status bits according to the comparison.

Status bits affected:

Logical greater than, arithmetic greater than, and equal.

```
 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
|L>|A>|EQ|C |OV|OP|X |--------------| INT. MASK |
 ^  ^  ^
```

Execution results:

(gas) => (gad)

Application notes:

The MOV instruction moves 16-bit words as follows:

Memory-to-memory (non-register)
Load register (memory-to-register)
Register-to-register
Register-to-memory

The MOV instruction may also be used to compare a memory location to zero. For example,

```
MOV     7,7
JNE     TEST
```

moves Workspace Register 7 to itself and compares the contents of Workspace Register 7 to zero. If the contents are not equal to zero, the equal status bit is reset and control transfers to TEST.

As another example of the use of MOV, assume that Workspace Register 9 contains >3416 and location ONES contains >FFFF. Then

```
MOV     @ONES,9
```

changes the contents of Workspace Register 9 to >FFFF, while the contents of location ONES is not changed. For this example, the logical greater than status bit is set and the arithmetic greater than and equal status bits are reset.

10.5 MOVE BYTE--MOVB

Op-code: D000 (Format I)

Syntax definition:

[<label>] b MOVB b (gas),(gad) b [<comment>]

Example:

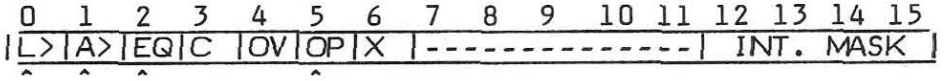
```
NEXT    MOVB    2,>2A41     Stores the most significant byte of
                            Workspace Register 3 in address >2A41.
```

Definition:

Replaces the destination operand (byte) with a copy of the source operand (byte). If either operand is addressed in the Workspace Register mode, the byte addressed is the most significant byte. The least significant byte is not affected. The computer compares the destination operand to zero and sets/resets the status bits to indicate the result of the comparison. The odd parity bit is set when the bits in the destination operand establish odd parity.

Status bits affected:

Logical greater than, arithmetic greater than, equal, and odd parity.



Execution results:

(gas) => (gad)

Application notes:

The MOVB instruction moves bytes in the same combinations as the MOV instruction moves words. For example, if memory location >1C14 contains a value of >2016 and TEMP is located at >1C15, and if Workspace Register 3 contains >542B, the instruction

```
MOVB    @TEMP,3
```

changes the contents of Workspace Register 3 to >162B. The logical greater than, arithmetic greater than, and odd parity status bits are set, while the equal status bit is reset.

10.6 STORE STATUS--STST

Op-code: 02C0 (Format VIII)

Syntax definition:

[<label>] b STST b (wa) b [<comment>]

Example:

```
LABEL   STST    7       Stores status in Workspace Register 7.
```

Definition:

Stores the Status Register contents in the specified Workspace Register.

Status bits affected:

None.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
L>	A>	EQ	C	OV	OP	X	-	-	-	-	-	INT.	MASK		

Execution results:

(ST) => (wa)

Application notes:

The STST instruction stores the Status Register in the specified Workspace Register.

10.7 STORE WORKSPACE POINTER--STWP

Op-code: 02A0 (Format VIII)

Syntax definition:

[<label>] b STWP b (wa) b [<comment>]

Example:

```
LABEL   STWP   6        Stores the Workspace Pointer in Workspace
                        Register 6.
```

Definition:

Places a copy of the Workspace Pointer contents in the specified Workspace Register.

Status bits affected:

None.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
L>	A>	EQ	C	OV	OP	X	----------					INT. MASK			

Execution results:

(WP) => (wa)

Application notes:

The STWP instruction stores the contents of the Workspace Pointer in the specified Workspace Register.

10.8 SWAP BYTES--SWPB

Op-code: 06C0 (Format VI)

Syntax definition:

[<label>] b SWPB b (gas) b [<comment>]

Example:

```
SWITCH   SWPB    3        Switchs the most significant and least significant
                          bytes in Workspace Register 3.
```

Definition:

Replaces the most significant byte (bits 0-7) of the source operand with a copy of the least significant byte (bits 8-15) of the source operand and replaces the least significant byte with a copy of the most significant byte.

Status bits affected:

None.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
L>	A>	EQ	C	OV	OP	X	-	-	-	-	-	INT.	MASK		

Execution results:

Exchanges left and right bytes of word (gas).

Application notes:

Use the SWPB instruction to interchange bytes of an operand prior to executing various byte instructions. For example, if Workspace Register 0 contains >2144 and memory location >2144 contains the value >F312, the instruction

```
SWPB    *0+
```

changes the contents of memory location >2144 to >12F3 and increments Workspace Register 0 to >2146. The Status Register is unchanged.

10.9 INSTRUCTION EXAMPLE

The following program segment illustrates the use of many of the instructions discussed in this section. The first six instructions are a portion of a program that calls a subroutine labeled SUBR. The calling program performs some initialization and then calls the subroutine to check the contents of a 20-byte buffer. If the subroutine finds that the buffer contains byte values that are in numerically sequential order, then it returns >00 to the calling program in Workspace Register 4. If the bytes are not in numerically sequential order, the subroutine returns >01 in Workspace Register 4. The program and subroutine are described in greater detail after the program is listed.

```
        LWPI    >AC20     Load Workspacer Pointer.
        BL      @SUBR     Call subroutine.
        DATA    BUFFER    Address of BUFFER.
        MOV     4,4       See if numbers are in sequence.
        JNE     NOSEQ     Jump if subroutine found non-sequential
                          numbers.
        .
        .
        .
SUBR    S       4,4       Clear Workspace Register 4.
        LI      10,20     Put loop count in Workspace Register 10.
        MOV     *11+,7    Put address of BUFFER in Workspace Register 7.
        MOVB    *7,6      Put first number in the left byte of Workspace
                          Register 6.
CHECK   MOV     *7+,8     Put two bytes in Workspace Register 8.
        SB      6,8       Check for sequence.
        JNE     OUT       Jump if out of sequence.
        AI      6,>100    Add one to sequence checker.
        SWPB    8         Put other byte in left half of register.
        SB      6,8       Check for sequence.
        JNE     OUT       Jump if out of sequence.
        AI      6,>100    Add one to sequence checker.
        DECT    10        Decrement loop counter.
        JGT     CHECK     Jump to check next two bytes.
        JMP     RETURN    Through checking, all in order.
OUT     INC     4         Set Workspace Register 4 to a non-zero value.
RETURN  B       *11       Return to calling program.
```

In the calling program, the LIMI instruction places a zero in the interrupt mask of the Status Register to turn off all maskable interrupts before loading the Workspace Pointer with the LWPI instruction and calling the subroutine.

The BL instruction transfers program control to the subroutine with the address following the BL instruction placed in Workspace Register 11 to allow for return to the program. The location following the BL instruction contains the address of the 20-byte buffer to be checked by the subroutine. The subroutine returns control to the MOV instruction in the calling program, which then checks to see if the subroutine found the bytes in numerically sequential order and jumps to location NOSEQ (not shown) if they were not.

The subroutine clears Workspace Register 4 with the S instruction and puts a loop counter value of 20 in Workspace Register 10 with the LI instruction.

Since Workspace Register 11 contains the address of the location following the BL instruction in the calling program, the MOV *11+,7 instruction copies the address of BUFFER into Workspace Register 7 and increments the address in Workspace Register 11 to the location following the DATA directive, setting the address to the MOV instruction for the return when the subroutine is finished. The MOVB *7,6 instruction copies the first byte value into the left byte of Workspace Register 6.

At label CHECK, the MOV instruction begins a loop that copies a word (two bytes) into Workspace Register 8 and auto-increments the address in Workspace Register 7 to the next word in the buffer. The left byte of Workspace Register 8 is subtracted from its right byte. A non-zero result indicates an out of sequence number and the JNE instruction transfers control to the instruction labeled OUT which places a >01 in Workspace Register 4.

If the subtraction produces a zero result, the INC 6 instruction increments the contents of Workspace Register 6 to the next byte to be checked. The following SWPB instructon swaps the bytes in Workspace Register 8 so the following SB and JNE instructions can check the sequence. If the sequence is correct, the next INC instruction updates Workspace Register 6 to the address of the next byte.

The DECT instruction decrements the loop counter in Workspace Register 10 by two since two bytes have been checked. If the result is non-zero, there are more bytes to be checked and the JGT instruction causes a reiteration of the loop. If the result is zero, all 20 bytes have been checked and the JMP instruction causes a jump to the subroutine's exit at RETURN. There the B *11 instruction causes a return to the calling program.

SECTION 11: LOGICAL INSTRUCTIONS

The following logical instructions are described in this section.

Instruction	Mnemonic	Section
AND Immediate	ANDI	11.1
OR Immediate	ORI	11.2
EXclusive OR	XOR	11.3
INVert	INV	11.4
CLeaR	CLR	11.5
SET to One	SETO	11.6
Set Ones Corresponding	SOC	11.7
Set Ones Corresponding, Byte	SOCB	11.8
Set Zeros Corresponding	SZC	11.9
Set Zeros Corresponding, Byte	SZCB	11.10

Logical instructions permit you to perform various logical operations on memory locations and/or Workspace Registers.

Each instruction consists of the following information.

- A *heading, consisting of the instruction name and mnemonic name*
- The op-code
- The syntax definition
- An example of the instruction
- The definition of the instruction
- The status bits affected
- The execution results
- Application notes when appropriate

The op-code is a four-digit hexadecimal number which corresponds to an instruction word whose address fields contain zeros.

The syntax definition follows the conventions described in Section 5. The generic names used in the syntax definitions are:

- gas General Address of the Source operand
- gad General Address of the Destination operand
- wa Workspace register Address
- iop Immediate OPerand
- wad Workspace register Address Destination
- disp DISPlacement of CRU lines from the CRU base register
- exp EXPression that represents an instruction location
- cnt CouNT of bits for CRU transfer
- scnt Shift CouNT
- xop number of eXtended OPeration

Source statements that contain machine instructions can use the label field, the operation field, the operand field, and the comment field.

Use of the label field is optional. When it is used, the label is assigned the address of the instruction. The Assembler advances to the location of a word boundary (even address) before assembling a machine instruction.

The operation (op-code) field contains the mnemonic operation code of the instruction. The contents of the operand field are defined for each instruction.

Inclusion of the comment field is optional. If used, it may contain any ASCII characters, including blanks. The comment has no effect on the assembly process other than being printed in the listing.

In the execution results, the following conventions are used.

- ( ) Indicates "the contents of."
- => Indicates "replaces."
- * * Indicates "the absolute value of."

The generic names used in the syntax definitions are also used in the execution results.

11.1 AND IMMEDIATE--ANDI

Op-code: 0240 (Format VIII)

Syntax definition:

[<label>] b ANDI b (wa),(iop) b [<comment>]

Example:

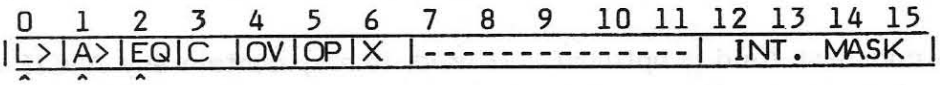
```
LABEL    ANDI    3,>FFF0      Sets least significant 4 bits of Workspace
                              Register 3 to zeros.
```

Definition:

Performs a bit-by-bit AND operation on the 16 bits in the immediate operand and the corresponding bits of the Workspace Register. The immediate operand is the word in memory immediately following the instruction word. Place the result in the Workspace Register. The computer compares the result to zero and sets/resets the status bits according to the results of the comparison.

Status bits affected:

Logical greater than, arithmetic greater than, and equal.



Execution results:

(wa) AND iop => (wa)

Application notes:

The ANDI instruction performs a logical AND with an immediate operand and a Workspace Register. Each bit of the 16-bit word of both operands follows the following table.

Immediate Operand Bit	Workspace Register Bit	AND Result
0	0	0
0	1	0
1	0	0
1	1	1

For example, if Workspace Register 0 contains >D2AB, the instruction

```
ANDI    0,>6D03
```

results in Workspace Register 0 changing to >4003. This AND operation on a bit-by-bit basis is

```
0110 1101 0000 0011  (Immediate operand--6D03)
1101 0010 1010 1011  (Workspace Register 0--D2AB)
---------------------
0100 0000 0000 0011  (Workspace Register 0 result--4003)
```

In this example, the logical greater than and arithmetic greater than status bits are set, while the equal status bit is reset.

11.2 OR IMMEDIATE--ORI

Op-code: 0260 (Format VIII)

Syntax definition:

[<label>] b ORI b (wa),(iop) b [<comment>]

Example:

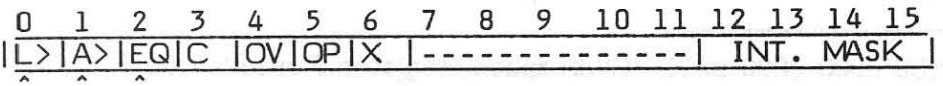
LABEL	ORI	3,>F000	Sets the most significant 4 bits of Workspace Register 3 to ones.

Definition:

Performs a logical OR operation on the 16-bit immediate operand and the corresponding bits of the Workspace Register. The immediate operand is the memory word immediately following the ORI instruction. Place the result in the Workspace Register. The computer compares the result to zero and sets/resets the status bits to indicate the result of the comparison.

Status bits affected:

Logical greater than, arithmetic greater than, and equal.



Execution results:

(wa) OR (iop) => (wa)

Application notes:

The ORI instruction performs a logical OR with the immediate operand and a specified Workspace Register. Each bit of the 16-bit word of both operands follows the following table.

Immediate Operand Bit	Workspace Register Bit	ORI Result
0	0	0
1	0	1
0	1	1
1	1	1

For example, if Workspace Register 5 contains >D2AB, the instruction

```
ORI     5,>6D03
```

results in Workspace Register 5 changing to >FFAB. This OR operation on a bit-by-bit basis is

```
0110 1101 0000 0011  (Immediate operand-->6D03)
1101 0010 1010 1011  (Workspace Register 5-->D2AB)
---------------------
1111 1111 1010 1011  (Workspace Register 5 result-->FFAB)
```

In this example, the logical greater than status bit is set, and the arithmetic greater than and equal status bit are reset.

11.3 EXCLUSIVE OR--XOR

Op-code: 2800 (Format III)

Syntax definition:

[<label>] b XOR b (gas),(wad) b [<comment>]

Example:

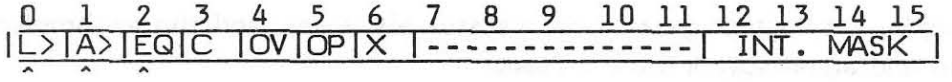
LABEL	XOR	@WORD,3	Exclusive ORs the contents of WORD and Workspace Register 3.

Definition:

Performs a bit-by-bit exclusive OR of the source and destination operands, and replaces the destination operand with the result. The exclusive OR is accomplished by setting the bits in the resultant destination operand to one when the corresponding bits of the two operands are not equal. The bits in the resultant destination operand are reset to zero when the corresponding bits of the two operands are equal. The computer compares the resultant destination operand to zero and sets/resets the status bits to indicate the result of the comparison.

Status bits affected:

Logical greater than, arithmetic greater than, and equal.



Execution results:

(gas) XOR (wad) => (wad)

Application notes:

The XOR instruction performs an exclusive OR on two word operands. Each bit of the 16-bit word of both operands follows the following table.

Immediate Operand Bit	Workspace Register Bit	XOR Result
0	0	0
0	1	1
1	0	1
1	1	0

For example, if Workspace Register 2 contains >D2AA and location CHANGE contains the value >6D03, the instruction

```
XOR     @CHANGE,2
```

results in the contents of Workspace Register 2 changing to >BFA9. Location CHANGE remains >6D03. This is shown as follows.

```
0110 1101 0000 0011  (Source operand-->6D03)
1101 0010 1010 1010  (Destination operand-->D2AA)
---------------------
1011 1111 1010 1001  (Destination operand result-->BFA9)
```

In this example, the logical greater than status bit is set, while the arithmetic greater than and equal status bits are reset.

11.4 INVERT--INV

Op-code: 0540 (Format VI)

Syntax definition:

[<label>] b INV b (gas) b [<comment>]

Example:

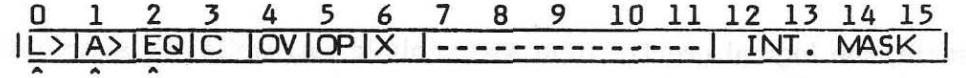
COMPL	INV	@BUFF(2)	Replaces the value at the address found by adding the value of Workspace Register 2 to the contents of BUFF with the one's complement of the data.

Definition:

Replaces the source operand with the one's complement of the source operand. The one's complement is equivalent to changing each zero in the source operand to one and each one in the source operand to zero. The computer compares the result to zero and sets/resets the status bits to indicate the result of the comparison.

Status bits affected:

Logical greater than, arithmetic greater than, and equal.



Execution results:

The one's complement of (gas) is placed in (gas).

Application notes:

The INV instruction changes each zero in the source operand to one and each one to zero. For example, if Workspace Register 11 contains >157A, the instruction

```
INV     11
```

changes the contents of Workspace Register 11 to >EA85. This INV operation on a bit-by-bit basis is

```
0001 0101 0111 1010  (Workspace Register 11-->157A)
1110 1010 1000 0101  (Workspace Register 11 result-->EA85)
```

The logical greater than status bit is set and the arithmetic greater than and equal status bits are reset.

11.5 CLEAR--CLR

Op-code: 04C0 (Format VI)

Syntax definition:

[<label>] b CLR b (gas) b [<comment>]

Example:

```
PRELM   CLR     @BUFF(2)    Clears the value at the address found by adding
                            the value of Workspace Register 2 to the
                            contents of BUFF.
```

Definition:

Replaces the source operand with a full 16-bit word of zeros.

Status bits affected:

None.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
L>	A>	EQ	C	OV	OP	X	----------					INT. MASK			

Execution results:

0 => (gas)

Application notes:

The CLR instruction sets a full, 16-bit, memory-addressable word to zero. For example, if Workspace Register 11 contains the value >2001, the instruction

```
CLR     *>B
```

results in the contents of memory locations >2000 and >2001 being set to 0. Workspace Register 11 and the Status Register are unchanged. Word operations, such as CLR, operate on the next lower address when an odd address is given as the operand.

11.6 SET TO ONE--SETO

Op-code: 0700 (Format VI)

Syntax definition:

[<label>] b SETO b (gas) b [<comment>]

Example:

```
LABEL   SETO   3      Sets Workspace Register 3 to >FFFF or negative
                      1.
```

Definition:

Replaces the source operand with a full 16-bit word of ones.

Status bits affected:

None.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
L>	A>	EQ	C	OV	OP	X	----------					INT. MASK			

Execution results:

>FFFF => <gas>

Application notes:

The SETO instruction initializes an addressable memory to a value of negative 1. For example, the instruction

```
        SETO   3
```

initializes Workspace Register 3 to a value of >FFFF. The contents of the Status Register are unchanged. This is a useful means of setting flag words.

11.7 SET ONES CORRESPONDING--SOC

Op-code: E000 (Format I)

Syntax definition:

[<label>] b SOC b (gas),(gad) b [<comment>]

Example:

LABEL	SOC	3,2	ORs Workspace Register 3 into Workspace Register 2.

Definition:

Sets to one the bits in the destination operand that correspond to the one bits in the source operand. Leaves unchanged the bits in the destination operand that are in the same bit positions as the zero bits in the source operand. This operation is an OR of the two operands. The changed destination operand replaces the original destination operand. The computer compares the result to zero and sets/resets the status bits to indicate the result of the comparison.

Status bits affected:

Logical greater than, arithmetic greater than, and equal.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
L>	A>	EQ	C	OV	OP	X	----------					INT. MASK			
^	^	^													

Execution results:

The bits of (gad) that correspond to the bits of (gas) that are equal to 1 are set to 1.

Application notes:

The SOC instruction ORs the 16-bit contents of two operands. For example, if Workspace Register 3 contains >FF00 and location NEW contains >AAAA, the instruction

```
SOC     3,@NEW
```

changes the contents of location NEW to >FFAA, while the contents of Workspace Register 3 are unchanged. This SOC operation on a bit-by-bit basis is

```
1111 1111 0000 0000  (Source operand-->FF00)
1010 1010 1010 1010  (Destination operand-->AAAA)
---------------------
1111 1111 1010 1010  (Destination operand result-->FFAA)
```

In this example, the logical greater than status bit is set and the arithmetic greater than and equal status bits are reset.

11.8 SET ONES CORRESPONDING, BYTE--SOCB

Op-code: F000 (Format I)

Syntax definition:

[<label>] b SOCB b (gas),(gad) b [<comment>]

Example:

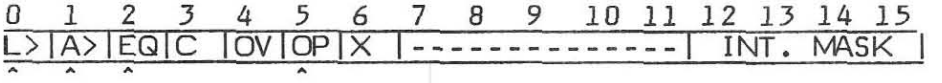
```
LABEL   SOCB   3,@DET      ORs Workspace Register 3 into the byte at
                           location DET.
```

Definition:

Sets to one the bits in the destination operand that correspond to the one bits in the source operand byte. Leaves unchanged the bits in the destination operand that are in the same bit positions as the zero bits in the source operand byte. This operation is an OR of the two operand bytes. The changed destination operand byte replaces the original destination operand byte. The computer compares the resulting destination operand byte to zero and sets/resets the status bits to indicate the results of the comparison. The odd parity status bit is set when the bits in the resulting byte establish odd parity.

Status bits affected:

Logical greater than, arithmetic greater than, equal, and odd parity.



Execution results:

The bits of (gad) that correspond to the bits of (gas) that are equal to 1 are set to 1.

Application notes:

The SOCB instruction ORs two byte operands. For example, if Workspace Register 5 contains >F013 and Workspace Register 8 contains the value >AA24, the instruction

```
SOCB    3,8
```

changes the contents of Workspace Register 8 to >FA24, while the contents of Workspace Register 5 are unchanged. This SOCB operation on a bit-by-bit basis is

```
1111 0000 0001 0011  (Source operand-->F013)
1010 1010 0010 0100  (Destination operand-->AA24)
---------------------
1111 1010 0010 0100  (Destination operand result-->FA24)
          ----------
          (Unchanged)
```

In this example, the logical greater than status bit is set, while the arithmetic greater than, equal, and odd parity status bits are reset.

11.9 SET ZEROS CORRESPONDING--SZC

Op-code: 4000 (Format I)

Syntax definition:

[<label>] b SZC b (gas),(gad) b [<comment>]

Example:

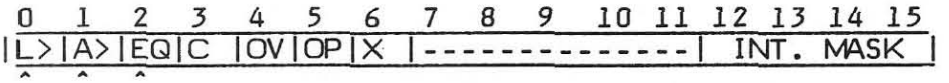
LABEL	SZC	@MASK,2	Resets the bits of Workspace Register 2 as indicated by MASK.

Definition:

Sets to zero the bits in the destination operand that correspond to the bit positions equal to one in the source operand. This operation is effectively an AND operation of the destination operand and the one's complement of the source operand. The computer compares the resulting destination operand to zero and sets/resets the status bits to indicate the results of the comparison.

Status bits affected:

Logical greater than, arithmetic greater than, and equal.



Execution results:

The bits of (gad) that correspond to the bits of (gas) that are equal to 1 are set to 0.

Application notes:

The SZC instruction turns off flag bits or ANDs the destination operand. For example, if Workspace Register 5 contains >6D03 and Workspace Register 3 contains >D2AA, the instruction

```
SZC     5,3
```

changes the contents of Workspace Register 3 to >92A8, while the contents of Workspace Register 5 remain unchanged. This SCZ operation on a bit-by-bit basis is

```
0110 1101 0000 0011  (Source operand-->6D03)
1101 0010 1010 1010  (Destination operand-->D2AA)
--------------------
1001 0010 1010 1000  (Destination operand result-->92A8)
```

In this example, the logical greater than status bit is set, while the arithmetic greater than and equal status bits are reset.

11.10 SET ZEROS CORRESPONDING, BYTE--SZCB

Op-code: 5000 (Format I)

Syntax definition:

[<label>] b SZCB b (gas),(gad) b [<comment>]

Example:

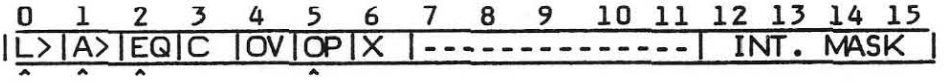
LABEL	SZCB	@MASK,@CHAR	Resets the bits of CHAR as indicated by MASK.

Definition:

Sets to zero the bits in the destination operand byte that correspond to the bit positions equal to one in the source operand byte. This operation is effectively an AND operation of the destination operand byte and the one's complement of the source operand byte. The computer compares the resulting destination operation to zero and sets/resets the status bits to indicate the results of the comparison. The odd parity status bit is set when the bits in the resulting destination operand byte establish odd parity. When the destination operand is given as a Workspace Register, the most significant byte is the one affected.

Status bits affected:

Logical greater than, arithmetic greater than, equal, and odd parity.



Execution results:

The bits of (gad) that correspond to the bits of (gas) that are equal to 1 are set to 0.

Application notes:

The SZCB instruction is used for the same applications as SZC except that bytes are used instead of words. For example, if location BITS contains the value >F018 and location TESTVA contains the value >AA24, the instruction

```
SZCB    @BITS,@TESTVA
```

changes the contents of TESTVA to >0A24, while BITS remains unchanged. This is shown as

```
1111 0000 0001 1000  (Source operand-->F018)
1010 1010 0010 0100  (Destination operand-->AA24)
---------------------
0000 1010 0010 0100  (Destination operand result-->0A24)
          ----------
          (Unchanged)
```

In this example, the logical greater than and arithmetic greater than status bits are set, while the equal and odd parity status bits are reset.

SECTION 12: WORKSPACE REGISTER SHIFT INSTRUCTIONS

The following Workspace Register shift instructions are described in this section.

Instruction	Mnemonic	Section
Shift Right Arithmetic	SRA	12.1
Shift Right Logical	SRL	12.2
Shift Left Arithmetic	SLA	12.3
Shift Right Circular	SRC	12.4

An example is given in Section 12.5.

Workspace Register shift instructions permit you to shift the contents of a specified Workspace Register from one to 16 bits. For each of these instructions, if the shift count in the instruction is zero, the shift count is taken from Workspace Register 0, bits 12 through 15. If the four bits of Workspace Register 0 are equal to zero, the shift count is 16 bit positions. The value of the last bit shifted out of the Workspace Register is placed in the carry bit of the Status Register. The result is compared to zero, and the results of the comparison are shown in the logical greater than, arithmetic greater than, and equal bits in the Status Register. If a shift count greater than 15 is supplied, the Assembler fills in the four-bit field with the least significant four bits of the shift count.

Each instruction consists of the following information.

- o A heading, consisting of the instruction name and mnemonic name
- o The op-code
- o The syntax definition
- o An example of the instruction
- o The definition of the instruction
- o The status bits affected
- o The execution results
- o Application notes when appropriate

The op-code is a four-digit hexadecimal number which corresponds to an instruction word whose address fields contain zeros.

The syntax definition follows the conventions described in Section 5. The generic names used in the syntax definitions are:

gas	General Address of the Source operand
gad	General Address of the Destination operand
wa	Workspace register Address
iop	Immediate OPerand
wad	Workspace register Address Destination
disp	DISPlacement of CRU lines from the CRU base register
exp	EXPression that represents an instruction location
cnt	CouNT of bits for CRU transfer
scnt	Shift CouNT
xop	number of eXtended OPeration

Source statements that contain machine instructions can use the label field, the operation field, the operand field, and the comment field.

Use of the label field is optional. When it is used, the label is assigned the address of the instruction. The Assembler advances to the location of a word boundary (even address) before assembling a machine instruction.

The operation (op-code) field contains the mnemonic operation code of the instruction. The contents of the operand field are defined for each instruction.

Inclusion of the comment field is optional. If used, it may contain any ASCII characters, including blanks. The comment has no effect on the assembly process other than being printed in the listing.

In the execution results, the following conventions are used.

( )	Indicates "the contents of."
=>	Indicates "replaces."
* *	Indicates "the absolute value of."

The generic names used in the syntax definitions are also used in the execution results.

12.1 SHIFT RIGHT ARITHMETIC--SRA

Op-code: 0800 (Format V)

Syntax definition:

[<label>] b SRA b (wa),(scnt) b [<comment>]

Example:

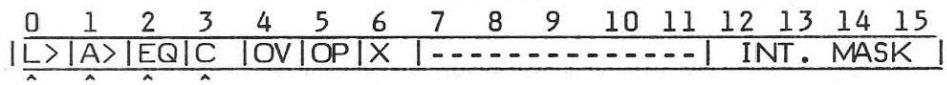
LABEL	SRA	2,3	Shifts Workspace Register 2 right 3 bit locations.

Definition:

Shifts the contents of the specified Workspace Register to the right for the specified number of bit positions. Fills vacated bit positions with the sign bit.

Status bits affected:

Logical greater than, arithmetic greater than, equal, and carry.



Execution results:

Shifts the bits of (wa) to the right, extending the sign bit to fill vacated bit positions. When scnt is greater than 0, shifts the number of bit positions specified by scnt. If scnt is equal to 0, shifts the number of bit positions contained in the four least significant bits of Workspace Register 0. If scnt and the four least significant bits of Workspace Register 0 both contain 0s, shifts 16 positions.

Application notes:

The SRA instruction shifts the given Workspace Register to the right the given number of bit positions and fills vacated positions with the sign bit. If Workspace Register 5 contains the value >8224, and Workspace Register 0 contains the value >F326, the instruction

```
SRA     5,0
```

changes the contents of Workspace Register 5 to >FE08. This SRA operation on a bit-by-bit basis is

```
1111 0011 0010 0110  (Workspace Register 0-->F326  Four least
                      significant bits are 0110, so shift 6 positions)
1000 0010 0010 0100  (Workspace Register 5-->8224)
---------------------
1111 1110 0000 1000  (Workspace Register 5 result-->FE08)
```

The logical greater than and carry status bits are set, while the arithmetic greater than and equal status bits are reset.

12.2 SHIFT RIGHT LOGICAL--SRL

Op-code: 0900 (Format V)

Syntax definition:

[<label>] b SRL b (wa),(scnt) b [<comment>]

Example:

```
LABEL   SRL     2,7         Shifts Workspace Register 2 right 7 bit
                            locations.
```

Definition:

Shifts the contents of the specified Workspace Register to the right the specified number of bits. Fills the vacated bit positions with zeros.

Status bits affected:

Logical greater than, arithmetic greater than, equal, and carry.

```
 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
|L>|A>|EQ|C |OV|OP|X |--------------| INT. MASK |
 ^  ^  ^  ^
```

Execution results:

Shifts the bits of (wa) to the right, filling the vacated bit positions with zeros. If scnt is greater than 0, shifts the number of bit positions specified by scnt. If scnt is equal to 0, shifts the number of bit positions contained in the four least significant bits of Workspace Register 0. If scnt and the four least significant bits of Workspace Register 0 both contain 0s, shifts 16 bit positions.

Application notes:

The SRL instruction shifts the given Workspace Register to the right the given number of bit positions and fills vacated positions with zeros. If Workspace Register zero contains the value >FFEF, the instruction

```
SRL     0,3
```

changes the contents of Workspace Register 0 to >1FFD. This SRL operation on a bit-by-bit basis is

```
1111 1111 1110 1111  (Workspace Register 0-->FFEF)
---------------------
0001 1111 1111 1101  (Workspace Register 0 result-->1FFD)
```

The logical greater than, arithmetic greater than and carry status bits are set, while the equal status bit is reset.

12.3 SHIFT LEFT ARITHMETIC--SLA

Op-code: 0A00 (Format V)

Syntax definition:

[<label>] b SLA b (wa),(scnt) b [<comment>]

Example:

```
LABEL    SLA    2,1          Shifts Workspace Register 2 left 1 bit location.
```

Definition:

Shifts the contents of the specified Workspace Register to the left the specified number of bit positions. Fills the vacated bit positions with zeros. Note that the overflow status bit is set when the sign of the word changes during the shift operation. The carry status bit contains the value shifted out of bit position zero.

Status bits affected:

Logical greater than, arithmetic greater than, equal, carry, and overflow.

```
 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
|L>|A>|EQ|C  |OV|OP|X  |--------------| INT. MASK |
 ^  ^  ^  ^  ^
```

Execution results:

Shifts the bits of (wa) to the left, filling the vacated bit positions with zeros. When scnt is greater than 0, shifts the number of bit positions specified by scnt. If scnt is equal to 0, shifts the number of bit positions contained in the four least significant bits of Workspace Register 0. If scnt and the four least significant bits of Workspace Register 0 both contain 0s, shifts 16 bit positions.

Application notes:

The SLA instruction shifts the given Workspace Register to the left the given number of bit positions and fills vacated positions with zeros. If Workspace Register 10 contains the value >1357, the instruction

```
SLA     10,5
```

changes the contents of Workspace Register 10 to >6AE0. This SLA operation on a bit-by-bit basis is

```
0001 0011 0101 0111  (Workspace Register 10-->1357)
---------------------
0110 1010 1110 0000  (Workspace Register 10 result-->6AE0)
```

The logical greater than, arithmetic greater than, and overflow status bits are set, while the equal and carry status bits are reset. Refer to Section 12.5 for another example.

12.4 SHIFT RIGHT CIRCULAR--SRC

Op-code: 0B00 (Format V)

Syntax definition:

[<label>] b SRC b (wa),(scnt) b [<comment>]

Example:

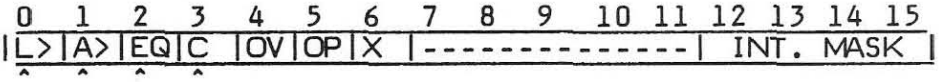
LABEL	SRC	7,16-3	Shifts Workspace Register 7 circularly 3 bit locations right.

Definition:

Shifts the specified Workspace Register to the right the specified number of bit positions. Fills vacated bit positions with the bit shifted out of position 15. The carry status bit contains the value shifted out of bit position zero.

Status bits affected:

Logical greater than, arithmetic greater than, equal, and carry.



Execution results:

Shifts the bits of (wa) to the right, filling the vacated bit positions with the bits shifted out at the right. If scnt is greater than 0, shifts the number of bit positions specified by scnt. If scnt is equal to 0, shifts the number of bit positions contained in the four least significant bits of Workspace Register 0. If scnt and the four least significant bits of Workspace Register 0 both contain 0s, shifts 16 bit positions.

Application notes:

The SRC instruction shifts the given Workspace Register to the right the given number of bit positions and fills vacated positons with the bits shifted. If Workspace Register 2 contains the value >FFEF, the instruction

```
SRC     2,7
```

changes the contents of Workspace Register 2 to >DFFF. This SRC operation on a bit-by-bit basis is

```
1111 1111 1110 1111  (Workspace Register 2-->FFEF)
---------------------
1101 1111 1111 1111  (Workspace Register 2 result-->DFFF)
```

The logical greater than and carry status bits are set, while the arithmetic greater than and equal status bits are reset.

There is no "shift left circular" instruction because the same effect can be obtained with SRC. To shift left a number of bits, instead shift right by a number equal to 16 minus the number. For example, to shift left 7 bits, shift right by 16 minus 7 or 9 bits.

12.5 INSTRUCTION EXAMPLE

This shift instruction shifts the indicated Workspace Register a specified number of bits to the left. For example, the instruction

```
SLA     5,1
```

shifts the contents of Workspace Register five one bit to the left. The carry status bit contains the value shifted out of bit position zero. The jump instructions JOC and JNC permit you to test the shifted bit. The overflow status bit is set when the sign of the contents of the Workspace Register being shifted changes during the shift operation. If Workspace Register 5 contains

```
0100 1111 0000 1111
```

before the shift, the instruction changes Workspace Register 5 to

```
1001 1110 0001 1110
```

The carry status bit contains a zero and the overflow status bit is set because the contents changed from positive to negative (bit zero changed from 0 to 1). If this shift sign change is significant, you could insert a JNO instruction to test the overflow condition. If there is no overflow, control transfers to the normal program sequence. Otherwise, the next instruction is executed.

It is possible to construct double-length shifts with the SLA instruction, to shift two or more words in a Workspace. The following code shifts two consecutive Workspace Registers assuming that:

- o The contents of Workspace Registers 1 and 2 are shifted one bit position.
- o Additional code could be included to execute the code once for each bit shift required, when shifts of more than one bit position are required. The additional code must include a means of testing that the desired number of shifts are performed.
- o Additional code tests for overflow from Workspace Register 1, to branch to an error routine at location ERR when overflow occurs.

```
        SLA     1,1         Shift R1 one bit.
        JOC     ERR
        SLA     2,1         Shift R2 one bit.
        JNC     EXIT        Transfer if no carry.
        INC     1           Transfer bit from R2 to R1.
        .
        .
        .
EXIT    INC     1           Continue with program.
        .
        .
        .
ERR     NOP
```

SECTION 13: PSEUDO-INSTRUCTIONS

A pseudo-instruction has the form of a machine instruction but its unusual characteristics do not allow it to be a machine instruction by proper definition. Each pseudo-instruction serves a unique purpose. The Assembler includes two pseudo-instructions which are discussed below.

Instruction	Mnemonic	Section
No OPeration	NOP	13.1
ReTurn	RT	13.2

13.1 NO OPERATION--NOP

Syntax definiton:

[<label>] b NOP b [<comment>]

NOP places a machine instruction in the object code which only effects the execution time of the program. Use of the label field is optional. When the label field is used, the label is assigned the location of the instruction. The operation field contains NOP. The operand field is not used. Use of the comment field is optional.

Enter the NOP pseudo-instructions as shown in the following example.

```
MOD     NOP
```

Location MOD contains a NOP pseudo-instruction when the program is loaded. Another instruction may be placed in location MOD during execution to implement a program option. The Assembler supplies the same object code as though the source statement contained the following code.

```
MOD     JMP     $+2
```

13.2 RETURN--RT

Syntax definition:

[<label>] b RT b [<comment>]

RT places a machine instruction in the object code to return control from a subroutine to a calling routine. Use of the label field is optional. When the label field is used, the label is assigned the location of the instruction. The operation field contains RT. The operand field is not used. Use of the comment field is optional.

Enter the RT pseudo-instruction as shown in the following example.

```
RT
```

The Assembler supplies the same object code as though the source statement contained the following code.

```
B       *11
```

When control is transferred to a subroutine by execution of a BL instruction, the link to the calling routine is stored in Workspace Register 11. An RT pseudo-instruction returns program control to the instruction following the BL instruction in the calling routine.

See Section 24.11 for more information.

SECTION 14: *ASSEMBLER DIRECTIVES*

You can do much to affect the assembly process by placing assembler directives in your source code. With these directives, you can

- Alter the Location Counter.
- Change Assembler output.
- Initialize constants.
- Provide linkage between programs.
- Define extended operations and end programs.

These classes of directives are discussed in this section in the order listed.

Each directive consists of the following information.

- The directive name and mnemonic name.
- The syntax definition, following the conventions described in Section 5.
- An *example of the directive.*
- The definition of the directive.
- Application notes when appropriate.

In Assembler directives, use of the label field is optional. When a label is used, it is assigned the address of the directive. Inclusion of the comment field is also optional. If used, it may contain any ASCII characters, including blanks. The comment has no effect on the assembly process other than being printed in the listing.

14.1 DIRECTIVES THAT AFFECT THE LOCATION COUNTER

As the Assembler reads the source statements of a program, a component of the Assembler, called the Location Counter, advances to correspond to the memory locations assigned to the resulting object code. The first five assembler directives discussed in this section (AORG, RORG, DORG, BSS, and BES) initialize the Location Counter and set up blocks of code, allowing you to place code at the location in memory where it will operate most efficiently when there are special memory requirements. RORG also forces object code to be relocatable so that the computer can place it where it is most efficient. EVEN places the Location Counter at an even word boundary.

PSEG and PEND are standard conditions, so they are not ordinarily used. With another loader, which allowed the use of CSEG, CEND, DSEG and DEND, they would be useful. The directives that affect the Location Counter and are useable with the Loader provided are shown below.

Directive	Mnemonic	Section
Absolute ORiGin	AORG	14.1.1
Relocatable ORiGin	RORG	14.1.2
Dummy ORiGin	DORG	14.1.3
Block Starting with Symbol	BSS	14.1.4
Block Ending with Symbol	BES	14.1.5
Word boundary	EVEN	14.1.6
Program SEGment	PSEG	14.1.7
Program segment END	PEND	14.1.8

Other directives, for setting up various kinds of segments, are also available for use with a loader which you provide. They are discussed in detail so that you may write a loader which uses them. They are not acceptable to the Loader provided. The directives that affect the Location Counter but are not useable with the Loader provided are shown below.

Directive	Mnemonic	Section
Common SEGment	CSEG	14.1.9
Common segment END	CEND	14.1.10
Data SEGment	DSEG	14.1.11
Data segment END	DEND	14.1.12

14.1.1 Absolute Origin--AORG

Syntax definition:

[<label>] b AORG b <wd-exp> b [<comment>]

Example:

```
LABEL   AORG   >C000+X
```

If X has a value of 6, the Location Counter is set to >C006. LABEL is assigned the value >C006.

Definition:

Places a value in the Location Counter and defines the following locations as absolute, enabling you to specify the exact locations in which object code is loaded. If a label is used, it is assigned the value that the directive places in the Location Counter. If you do not include an AORG directive in your program, the object code contains no absolute addresses.

14.1.2 Relocatable Origin--RORG

Syntax definition:

[<label>] b RORG b [<exp>] b [<comment>]

Example:

```
LABEL   RORG   $-20
```

Overlays 10 words. The $ symbol refers to the location following the preceding relocatable location of the program. This example backs up the Location Counter 10 words. LABEL is assigned the value placed in the Location Counter.

Definition:

Places a value in the Location Counter which, if encountered in absolute code, also defines succeeding locations as program-relocatable. If a label is used, it is assigned the value that the directive places into the Location Counter. The operation field contains RORG, and the operand field is optional. The comment field can be used only when the operand field is used.

If the operand field is not used, the length of the program segment, data segment, or specific common segment of a program replaces the value of the Location Counter. For a given relocation type X, the length of the X-relocatable segment is zero if no program-relocatable code has been previously assembled. Otherwise, it is the maximum value of the Location Counter due to the assembly of any preceding block of X-relocatable code.

Since the Location Counter begins at zero, the length of a segment and the "next available" address within that segment are identical.

If the RORG directive appears in absolute or relocatable code and the operand field is not used, the Location Counter value is replaced by the current length of the program segment of that program. If the directive appears in data-relocatable code without an operand, the Location Counter value is replaced by the length of the data segment. Similarly, in common-relocatable code a RORG directive without an operand causes the length of the appropriate common segment to be loaded into the Location Counter.

If the operand field is used, the operand must be an absolute or relocatable expression (exp) containing only previously defined symbols. If the directive is in absolute code, a relocatable operand must match the current Location Counter. If the RORG directive appears in absolute code, it changes the Location Counter to program-relocatable and replaces its value with the operand value. In relocatable code, the operand value replaces the current Location Counter value, and the relocation type of the Location Counter remains unchanged.

Application notes:

You may use the RORG directive to replace previous instructions and directives. This is the purpose of the example. Alternatively, the RORG directive can be used without an operand field.

For example, suppose your program starts with statements defining data which occupies >44 bytes, followed by an AORG directive, a BSS directive, statements in a block, and a BES directive to end the block. Then the directive

```
SEG2    RORG
```

places >0044 in the Location Counter and defines the Location Counter as relocatable. The symbol SEG2 is given a relocatable value of >0044. The RORG directive, as used here, has no effect except at the end of an absolute block or a dummy block.

14.1.3 Dummy Origin--DORG

Syntax definition:

[<label>] b DORG b <exp> b [<comment>]

Example:

```
LABEL   DORG   0
```

Causes the Assembler to assign values to the labels within the dummy section relative to the start of the dummy section.

Definition:

Places a value in the Location Counter and defines the following address locations as a dummy block or section. When assembling a dummy section, the Assembler does not generate object code but operates normally in all other respects. The result is that the symbols that describe the layout of the dummy section are available to the Assembler during assembly of the remainder of the program. The label is assigned the value that the directive places in the Location Counter. The operand field contains an expression which may be either absolute or relocatable. Any symbol in the expression must have been previously defined. If the operand is absolute, the Location Counter contents are absolute. If the operand is relocatable, the Location Counter contents are relocatable.

14.1.4 Block Starting with Symbol--BSS

Syntax definition:

[<label>] b BSS b <wd-exp> b [<comment>]

Example:

```
BUFF1   BSS    80
```

Reserves a 80-byte buffer at location BUFF1. BUFF1 is set equal to the previous value of the Location Counter.

Definition:

Advances the Location Counter by the value of the well-defined expression in the operand field. If a label is used, it is assigned the value of the location of the first byte in the block.

Application notes:

The BSS directive is used to start a block. Blocks are used to set up areas of code that you wish to have loaded into specific memory locations; for example, to set up a reference table. The AORG directive must precede the BSS directive.

14.1.5 Block Ending with Symbol--BES

Syntax definition:

[<label>] b BES b <wd-exp> b [<comment>]

Example:

BUFF2	BES	>10	Reserves a 16-byte buffer. If the Location Counter contains >100 when the Assembler processes this directive, BUFF2 is assigned the value >110.

Definition

Advances the Location Counter according to the value of the well-defined expression in the operand field. If a label is included, the directive assigns the new Location Counter value to the symbol in the label field. The BES directive marks the end of a block started with the BSS directive.

14.1.6 Word Boundary--EVEN

Syntax definition:

[<label>] b EVEN b [<comment>]

Example:

WRF1	EVEN	Assigns the Location Counter address to label WRF1 and ensures that the Location Counter contains a word boundary address.

Definition:

Places the Location Counter on the next word boundary (even) byte address. If the Location Counter is already on a word boundary, the Location Counter is not

altered. If a label is used, the value in the Location Counter is assigned to the label before processing the directive. The operand field is not used.

Application notes:

The EVEN directive ensures that the program is at an even word boundary when a statement that consists of only a label is preceded by a TEXT or BYTE directive and is followed by a DATA directive or a machine instruction. In this case, the label does not have the same value as a label in the following instruction unless the TEXT or BYTE directive left the Location Counter on an even (word) location.

Using an EVEN directive before or after a machine instruction or a DATA directive is redundant since the Assembler automatically advances the Location Counter to an even address when it processes a machine instruction or a DATA directive.

14.1.7 Program Segment--PSEG

Syntax definition:

[<label>] b PSEG b [<comment>]

Example:

```
LABEL    PSEG
```

Definition:

Places a value in the Location Counter and defines successive locations as program-relocatable. If a label is used, it is assigned the value that the directive places in the Location Counter. The value placed in the Location Counter as a result of this directive is zero if no program-relocatable code has been previously assembled. Otherwise, it is the maximum value the Location Counter has attained as a result of the assembly of any preceding block of program-relocatable code.

Application notes:

The PSEG directive only repeats the default mode. If you are using another loader that also accepts the CSEG, CEND, DSEG, and DEND directives, when the PSEG directive is useful.

14.1.8 Program Segment End--PEND

Syntax definition:

[<label>] b PEND b [<comment>]

Example:

```
LABEL    PEND
```

Definition:

Places a value in the Location Counter and defines the following locations as program-relocatable. If a label is used, it is assigned the value of the Location Counter prior to modification. The value placed in the Location Counter by this directive is the maximum value ever attained by the Location Counter as a result of the assembly of all preceding program-relocatable code. If the PEND directive is encountered in data-relocatable or common-relocatable code, it functions as a DEND or CEND and a warning message is issued. Like DEND and CEND, PEND is invalid if used in absolute code.

Application notes:

The PEND directive only repeats the default mode. If you are using another loader that also accepts the CSEG, CEND, DSEG, and DEND directives, then the PEND directive is useful. The PEND directive is provided as the program-segment counterpart to the DEND and CEND directives. However, since PEND properly appears only in program-relocatable code, the relocation type of successive locations remains unchanged.

14.1.9 Common Segment--CSEG

The CSEG directive is not accepted by the Loader provided with the Editor/Assembler.

Syntax definition:

[<label>] b CSEG b [<comment>]

Example:

```
COM1    CSEG
```

Definition:

If you provide an appropriate loader, CSEG places a value in the Location Counter and defines successive locations as common-relocatable (i.e., relocatable with respect to a common segment). If a label is used, it is assigned the value that the directive places in the Location Counter.

The CSEG directive defines the beginning (or continuation) of the "blank common" segment of the program.

Application notes:

The CSEG directive is not accepted by the Loader provided with the Editor/Assembler. The CEND, PSEG, DSEG, AORG, and END directives all terminate the definition of a block of common-relocatable code. The block is normally terminated with a CEND directive. Like CEND, the PSEG directive indicates that successive locations are program-relocatable. The DSEG and AORG directives effectively terminate the common segment by beginning a data segment or absolute segment. The END directive terminates the common segment as well as the program.

14.1.10 Common Segment End--CEND

The CEND directive is not accepted by the Loader provided with the Editor/Assembler.

Syntax definition:

[<label>] b CEND b [<comment>]

Example:

```
LABEL    CEND
```

Definition:

If you provide an appropriate loader, CEND terminates the definition of a block of common-relocatable code by placing a value in the Location Counter and defining successive locations as program-relocatable. If a label is used, it is assigned the value of the Location Counter prior to modification. The value placed in the Location Counter as a result of this directive is zero if no program-relocatable code has been previously assembled. Otherwise, it is the

maximum value the Location Counter has attained as a result of the assembly of any preceding block of program-relocatable code.

Application notes:

The CEND directive is not accepted by the Loader provided with the Editor/Assembler. If the directive is encountered in common-relocatable or program-relocatable code, it functions as a DEND or PEND directive and a warning message is issued. Like DEND and PEND, CEND is invalid if it is used in absolute code.

14.1.11 Data Segment--DSEG

The DSEG directive is not accepted by the Loader provided with the Editor/Assembler.

Syntax definition:

[<label>] b DSEG b [<comment>]

Example:

```
LABEL    DSEG
```

Definition:

If you provide an appropriate loader, DSEG places a value in the Location Counter and defines successive locations as data-relocatable. If a label is used, it is assigned the data-relocatable value that the directive places in the Location Counter. The value placed in the Location Counter as a result of this directive is zero if no program-relocatable code has been previously assembled. Otherwise, it is the maximum value the Location Counter has attained as a result of the assembly of any preceding block of program-relocatable code.

The DSEG directive defines the beginning of a block of data-relocatable code. The block is normally terminated with a DEND directive. If several such blocks appear throughout the program, they comprise the data segment of the program. The entire data segment can be relocated independently of the program segment when the programs are linked and therefore provides a convenient means of separating modifiable data from executable code.

Application notes:

The DSEG directive is not accepted by the Loader provided with the Editor/Assembler. In addition to the DEND directive, the PSEG, CSEG, AORG, and END directives properly terminate the definition of a block of data-relocatable code. Like DEND, the PSEG directive indicates that successive locations are program-relocatable. The CSEG and AORG directives effectively terminate the data segment by beginning a common segment or absolute segment, respectively. The END directive terminates the data segment as well as the program.

The following example illustrates the use of both the DSEG and the DEND directives.

```
RAM     DSEG        Start of data area.
          .
          .
          .
* Data-relocatable code.
          .
          .
          .
ERAM    DEND
*
LRAM    EQU         ERAM-RAM
```

The block of code between the DSEG and DEND directives is data-relocatable. RAM is the symbolic address of the first word of this block. ERAM is the data-relocatable byte address of the location following the code block. The value of the symbol LRAM is the length of the block in bytes.

14.1.12 Data Segment End--DEND

The DEND directive is not accepted by the Loader provided with the Editor/Assembler.

Syntax definition:

[<label>] b DEND b [<comment>]

Example:

```
LABEL    DEND
```

Definition:

If you provide an appropriate loader, DEND terminates the definition of a block of data-relocatable code by placing a value in the Location Counter and defining successive locations as program-relocatable. If a label is used, it is assigned the value of the Location Counter prior to modification. The value placed in the Location Counter as a result of this directive is zero if no program-relocatable code has been previously assembled. Otherwise, it is the maximum value the Location Counter has ever attained as a result of the assembly of any preceding block of program-relocatable code.

Application notes:

The DEND directive is not accepted by the Loader provided with the Editor/Assembler. If the directive is encountered in common-relocatable or program-relocatable code, it functions as a CEND or PEND and a warning message is issued. Like CEND and PEND, DEND is invalid if used in absolute code.

14.2 DIRECTIVES THAT AFFECT ASSEMBLER OUTPUT

In order to make Assembler output as easy to read as possible, you may specify a variety of output forms for the Assembler. These options include whether to list the source code, where to put pages, the page title to be used, and the program identifier to be used. In addition, several options are available when you select the Assembler. Refer to Section 2 for more details on the other output options. The directives that affect Assembler output are shown below.

Directive	Mnemonic	Section
No source List	UNL	14.2.1
LIST source	LIST	14.2.2
PAGE eject	PAGE	14.2.3
Page TITLe	TITL	14.2.4
Program IDenTifier	IDT	14.2.5

14.2.1 No Source List--UNL

Syntax definition:

[<label>] b UNL b [<comment>]

Example:

```
LABEL   UNL
```

Definition:

Stops printing of the source listing. The UNL directive is not printed in the source listing, but the line counter is incremented. If a label is used, the current value of the Location Counter is assigned to the label. The comment field is optional, but the Assembler does not print the comment. The UNL and LIST directives have no effect unless you have selected an output device and the L option has been selected. See Section 2 for more information.

Application notes:

The UNL directive stops the printing and thus reduces assembly time, as well as the size of the source listing.

14.2.2 List Source--LIST

Syntax definition:

[<label>] b LIST b [<comment>]

Example:

```
LABEL    LIST
```

Definition:

Restarts printing of the source listing. This directive is required only if you previously gave a No Source List (UNL) directive to cancel listing. The directive is not printed in the source listing, but the line counter is incremented. If a label is used, the current value of the Location Counter is assigned to the label. The comment field is optional, but the Assembler does not print the comment. The UNL and LIST directives have no effect unless you have selected an output device and the L option has been selected. See Section 2 for more information.

14.2.3 Page Eject--PAGE

Syntax definition:

[<label>] b PAGE b [<comment>]

Example:

```
LABEL    PAGE
```

Definition:

Causes the Assembler to continue the source program listing on a new page. The PAGE directive is not printed in the source listing, but the line counter is incremented. If a label is used, the current value of the Location Counter is assigned to the label. Use of the comment field is optional, but the Assembler does not print the comment.

Application notes:

The PAGE directive causes the Assembler to begin a new page of the source listing. The next source statement is the first statement listed on the new page. Use of the page directive to begin new pages of the source listing at the logical divisions of the program improves documentation of the program.

14.2.4 Page Title--TITL

Syntax definition:

[<label>] b TITL b '<string>' b [<comment>]

Example:

LABEL	TITL	'*REPORT GENERATOR*'	Causes the title *REPORT GENERATOR* to be printed in the subsequent page headings of the source listing.

Definition:

Supplies a title to be printed in the heading of the source listing. The directive is not printed in the source listing. If a label is used, the current value of the Location Counter is assigned to the label. The operand field contains the title as a string of up to 50 characters enclosed in single quotes. If you enter more than 50 characters, the Assembler retains the first 50 characters as the title and prints the message OUT OF RANGE. The Assembler does not print a comment included with the directive, but does increment the line counter.

To include a title in the heading of the first page of the source listing, a TITL directive must be the first source statement in your program. Then the title is printed on all pages until another TITL directive is processed. If the TITLE directive is not the first source statement, the title is printed on the next page after the directive is processed and on subsequent pages until another TITL directive is processed.

14.2.5 Program Identifier--IDT

Syntax definition:

[<label>] b IDT b '<string>' b [<comment>]

Example:

LABEL	IDT	'CONVERT'	Assigns the name CONVERT to the program to be assembled.

Definition:

Assigns a name to the program. If you use the IDT directive, it should precede any machine instruction or assembler directive that results in object code. If a label is used, the current value of the Location Counter is assigned to the label. The operand field contains the program name as a string of up to eight characters enclosed in single quotes. If more than eight characters are entered, the Assembler prints a truncation error message and retains the first eight characters as the program name.

The program name is printed in the source listing as the operand of the IDT directive but does not appear in the page heading of the source listing. The program name is placed in the object code but serves no purpose during the Assembly.

14.3 DIRECTIVES THAT INITIALIZE CONSTANTS

You may define the values of constants and place values in bytes and words with directives. The directives that initialize constants are shown below.

Directive	Mnemonic	Section
Define assembly-time constant	EQU	14.3.1
Initialize BYTE	BYTE	14.3.2
Initialize word	DATA	14.3.3
Initialize TEXT	TEXT	14.3.4

14.3.1 Define Assembly-Time Constant--EQU

Syntax definition:

<label> b EQU b <exp> b [<comment>]

Example:

```
BUFFER   EQU    >1000      Assigns the value >1000 to BUFFER.
```

Definition:

Assigns a value to a symbol. The label field contains the symbol. The operand field contains an expression in which all symbols have been previously defined. If a symbol is used in the operand field, and that symbol appears in the label field of a machine instruction in a relocatable block of the program, the value is relocatable. After the execution of this directive, the symbol in the label field and the value or symbol in the operand field may be used interchangeably.

14.3.2 Initialize Byte--BYTE

Syntax definition:

[<label>] b BYTE b <exp>[,<exp>]... b [<comment>]

Example:

```
KONS    BYTE >F+1,-1,0,'AB'-'AA'
```

Initializes four bytes, starting with the byte at location KONS. The contents of the resulting bytes are >10, >FF, >00, and >01.

Definition:

Places one or more values in one or more successive bytes of memory. If a label is used, the location at which the Assembler places the first byte is assigned to the label. The operand field contains one or more expressions separated by commas. The expressions must contain no external references. The Assembler evaluates each expression and places the value in a byte as an eight-bit two's complement number. If truncation is required, the Assembler prints a truncation error message and places the rightmost portion of the value in the byte. The EVEN directive is commonly used after the TEXT directive to insure that the next instruction starts on an even word boundary.

14.3.3 Initialize Word--DATA

Syntax definition:

[<label>] b DATA b <exp>[,<exp>]... b [<comment>]

Example:

```
KONS1   DATA    3200,1+'AB',-'AF','A'
```

Initializes four words, starting on a word boundary at location KONS1. The contents of the resulting words are >0C80, >4143, >BEBA, and >0041.

Definition:

Places one or more values in one or more successive words of memory. The Assembler automatically advances the Location Counter to a word boundary (even) address if necessary and places >00 in the byte skipped. If a label is used, the location at which the Assembler places the first word is assigned to the label. The operand field contains one or more expressions separated by commas. The Assembler evaluates each expression and places the value in a word as a 16-bit two's complement number.

14.3.4 Initialize Text--TEXT

Syntax definition:

[<label>] b TEXT b [-]'<string>' b [<comment>]

Example:

MSG1	TEXT	'EXAMPLE'	Places the seven ASCII hexadecimal representations of the characters in EXAMPLE in successive bytes. If the Location Counter is on an even address, the result is >4558, >414D, >504C, and >45XX where XX is determined by the next source statement. The label MSG1 is assigned the value of the first byte address.

Definition:

Places one or more characters in successive bytes of memory. The Assembler negates the last character of the string if the string is preceded by a unary minus (-) sign. If a label is used, the location at which the Assembler places the first character is assigned to the label. The operand field contains a character string of up to 52 characters, which can be preceded by a unary minus sign. The EVEN directive is commonly used after the TEXT directive to insure that the next instruction starts on an even word boundary.

14.4 DIRECTIVES THAT LINK PROGRAMS

It is often convenient to write programs as separate modules which can be linked together. Several directives allow you to do so. The DEF and REF directives allow you to place one or more symbols defined in a module into the object code, making them available for linking. The COPY directive allows you to have the Assembler copy a file from a diskette and include it in the assembly process.

The LOAD and SREF directives assemble properly but are not used by the Loader provided. They are discussed in detail so that you may write a loader which uses them. With an appropriate loader, different from the one provided, the LOAD and SREF directives allow you to place in the object code symbols used in the module but defined in another module, so that they can be linked. The directives that link programs are shown below.

Directive	Mnemonic	Section
External DEFinition	DEF	14.4.1
External REFerence	REF	14.4.2
COPY	COPY	14.4.3
Force LOAD	LOAD	14.4.4
Secondary REFerence	SREF	14.4.5

14.4.1 External Definition--DEF

Syntax definition:

[<label>] b DEF b <symbol>[,<symbol>]... b [<comment>]

Example:

LABEL	DEF	ENTR,ANS	Causes the Assembler to include symbols ENTR and ANS in the object code so that these symbols are available to other programs.

Definition:

Makes one or more symbols available to other programs for reference. If a label is used, the current value of the Location Counter is assigned to the label. The operand field contains one or more symbols, separated by commas, to be defined in the program being assembled. The DEF directive for a symbol must precede the source statements that contain the symbols, or the Assembler identifies the symbols as having been defined more than once and issues a duplicate definition warning message.

Application notes:

Labels that have been defined with the DEF directive are entered into the REF/DEF table and maintained in the REF/DEF table like other symbols defined by the DEF statement. Labels defined with the REF statement are resolved at loading time and removed fom the REF/DEF table.

Duplicate definitions are accepted by the Loader, with the most recent definition being used. A warning message is issued when duplicate definitions are given.

14.4.2 External Reference--REF

Syntax definition:

[<label>] b REF b <symbol>[,<symbol>] ... b [<comment>]

Example:

LABEL	REF	ARG1,ARG2	Causes the Assembler to include symbols ARG1 and ARG2 in the object code so that the corresponding addresses may be obtained from other programs.

Definition:

Provides access to one or more symbols defined in other programs. If a label is used, the current value of the Location Counter is assigned to the label. The operand field contains one or more symbols, separated by commas, to be used in the operand field of a subsequent source statement.

Application notes:

If a symbol is listed in the REF statement, a corresponding symbol must be present in a DEF statement in another source module. If a one-to-one matching of symbols does not occur, the error code >0D is given when the program is executed.

If a symbol in the operand field of a REF directive is the first operand of a DATA directive, the Assembler places the value of the symbol at location 0 of the routine. If that routine is loaded at absolute location 0, the symbol is not linked correctly. Use of the symbol at other locations is correctly linked.

14.4.3 Copy File--COPY

Syntax definition:

[<label>] b COPY b "<file name>" b

Example:

LABEL	COPY	"DSK1.MAIN"	Copies the file MAIN from the diskette in Disk Drive 1 for inclusion in the assembly process.

Definition:

Includes a file from a diskette in the assembly process. When the Assembler encounters this directive, it copies the file from the diskette and continues with the assembly process as if the file were in the program actually being assembled. You can include as many COPY directives as you wish in a program. Note, however, that when an END directive is encountered in any file, including those being copied, the Assembler stops the assembly process.

Application notes:

The COPY directive allows you to write programs as separate modules which can be linked together. This may be done for writing convenience or because the program is too large to fit in one file.

The following is a sample program which uses the COPY directive. When it is assembled, the Assembler first assembles the file named MAIN on Disk Drive 1, followed by the files PROG and DATA, also on Disk Drive 1. Then the rest of the program is assembled. The object file created includes all of the assembled files. The END statement tells the Assembler to stop assembling.

```
        COPY    "DSK1.MAIN"
        COPY    "DSK1.PROG"
        COPY    "DSK1.DATA"
        .
        .
        .
        END
```

14.4.3.1 Using the COPY Directive in the Game

In addition to the SAVE utility (See Section 24.5), the Editor/Assembler diskette labeled Part B contains a game or application program which uses the COPY directive.

The owner's manual for the the game or application program you received is included with your Editor/Assembler. You can also use your Disk Manager Command Module to catalog the diskette to find which game you received. The instructions below relate specifically to the game Tombstone City, but they apply generally to whichever game or application program you received.

TOMBS is the main module, consisting mostly of COPY directives. When it is assembled, the COPY directives copy the rest of the program files, which are TOMBA, TOMBB, TOMBC, TOMBD, TOMBE, and TOMBF. The program is broken into these portions because of its length.

Because the source listing, TOMB, is already on the diskette, the game may be played without further assembly. However, if you wish to assemble the file in order to see the process, first insert the diskette which contains the Assembler (lableled Part A) in Disk Drive 1 and choose the ASSEMBLE option on the Editor/Assembler. Then insert the diskette (labeled Part B) that contains the game in Disk Drive 1 and give the source file name as DSK1.TOMBS and the object file as DSK1.TOMB. If you have an RS232 Interface unit and an RS232-compatible printer, give the list file as RS232 and

options as LRSC. You cannot list to diskette because the file is too large. If you do not have an RS232 Interface unit, do not give a list file and assemble with options RC.

To run the program, select the LOAD AND RUN option on the Editor/Assembler. The file name is DSK1.TOMB. The program ends with the END START directive, so it begins running as soon as it is loaded. Alternatively, you may run the game from TI BASIC by running the following program:

```
100 CALL LOAD("DSK1.TOMB")
```

You may also alter the game so that it may be used with the RUN PROGRAM FILE option on the Editor/Assembler selection list. See Section 24.5 for instructions.

14.4.4 Force Load--LOAD

The LOAD directive is not accepted by the Loader provided with the Editor/Assembler.

Syntax definition:

[<label>] b LOAD b <symbol>[,<symbol>]... b [<comment>]

Example:

```
LABEL    LOAD    SYMBOL
```

Definition:

If you provide an appropriate loader, the LOAD directive loads a symbol into the REF/DEF table for later resolution. The LOAD directive is similar to REF, except that the symbol does not need to be used in the module containing the LOAD directive. The symbol included in the LOAD directive must be defined with the DEF directive in some other module. The LOAD directive is used together with the SREF directive. If a one-to-one matching of LOAD-SREF pairs and DEF symbols does not occur, unresolved references occur during linking.

14.4.5 Secondary External Reference--SREF

The SREF directive is not accepted by the Loader provided with the Editor/Assembler.

Syntax definition:

[<label>] b SREF b <symbol>,[<symbol>]... b [<comment>]

Example:

LABEL	SREF	ARG1,ARG2	Includes symbols ARG1 and ARG2 in the object code so that the corresponding addresses may be obtained from other programs.

Definition:

If you provide an appropriate loader, SREF provides access to one or more symbols defined in other programs. If a label is used, the current value of the Location Counter is assigned to the label. The operand field contains one or more symbols, separated by commas, to be used in the operand field of a subsequent source statement.

Application notes:

The SREF directive is not accepted by the Loader provided with the Editor/Assembler. SREF, unlike REF, does not require that a symbol have a corresponding symbol listed in a DEF statement of another source module. However, without a corresponding symbol, the symbol in the SREF directive is an unresolved reference.

14.5 MISCELLANEOUS DIRECTIVES

You may use the DXOP directive to define extended operations for use with the XOP instruction, which is only available on the TI-99/4A Home Computer. The END directive signals the end of a program and, when used with a label, starts execution of a program as soon as it is loaded. The miscellaneous directives are shown below.

Directive	Mnemonic	Section
Define eXtended OPeration	DXOP	14.5.1
Program END	END	14.5.2

14.5.1 Define Extended Operation--DXOP

Syntax definition:

[<label>] b DXOP b <symbol>,<term> b [<comment>]

Example:

```
LABEL   DXOP   DADD,1
```

Defines DADD as extended operation 1. When you include the symbol DADD in the operand field of an XOP instruction, the Assembler assembles an XOP instruction that specifies extended operation 1.

Definition:

The DXOP directive is only useful on the TI-99/4A Home Computer. To find which extended operations your computer supports, see Section 7.19. If available, the DXOP directive assigns a symbol to be used in the operator field to specify an extended operation. If a label is used, the current value in the Location Counter is assigned to the label. The operand field contains a symbol followed by a comma and a term. The symbol assigned to an extended operation must not be used in the label or operand field of any other statement. The Assembler assigns the symbol to an extended operation specified by the term, which has a value of 1 or 2.

14.5.2 Program End--END

Syntax definition:

[<label>] b END b [<symbol>] b [<comment>]

Example:

NAME	END	START	Terminates the assembly of the program. The Assembler also places the value of START in the object code as an entry point.

Definition:

Terminates the assembly process. The last source statement of a program is the END directive. Any source statements which follow the END directive are ignored. If a label is used, the current value in the Location Counter is assigned to the symbol. If the operand field is used, it contains a program-relocatable or absolute symbol that specifies the entry point of the program. The comment field may be used only when the operand field is used. If the END statement has an operand, the program runs as soon as it is loaded, starting at the address of the operand.

Application notes:

If the END statement is omitted, the Assembler issues an END ASSUMED warning message, and I/O error >07 is displayed. However, this should not cause a problem in loading and running the program.

If the operand field is not used, run the program by entering a program name when using the LOAD AND RUN option on the Editor/Assembler selection list or the CALL LINK statement in TI BASIC. The program name must be an entry point defined in the DEF instruction.

SECTION 15: ASSEMBLER OUTPUT

The major purpose of the assembly process is the production of object code so that your program can be run by the computer. This section includes a description of the object code produced so that you can edit it to make minor changes in your program.

In addition to object code, the Assembler prints a source listing and a sorted symbol table. You can affect the form of the listing by use of various directives discussed in Section 14.2. In addition, the Assembler produces a list of fatal and nonfatal errors. The purpose of all of this output is to enable you to discover errors in your program so that you can, by changing the source code, make the program run correctly.

15.1 SOURCE LISTING

The source listing shows the source statements and where their resulting object code is placed in memory. However, for the string following a TEXT directive, only the ASCII code for the first character is listed. A complete example is given in Section 15.5.

99/4 ASSEMBLER appears on the first line of each page. The second line contains VERSION 1.1 (or the version you are using), the title you supplied in a TITL directive, and the page number.

Below this heading material, the printer lists a line for each source statement containing the source statement number, the Location Counter value, the object code assembled (in hexadecimal notation), and the source statement as you entered it in your program. When a source statement generates more than one word of object code, the Assembler prints the Location Counter value and object code for each additional word of object code on a separate line following the source statement. An example of the material printed is

```
0018 0156 C820     MOV  @INIT+3,@3
     0158 012B'
     015A 0003
```

The source statement number, 0018 in the example, is a four-digit decimal number. Source records are numbered in the order in which they are entered, whether or not they are listed. The TITL, LIST, UNL, and PAGE directives are not listed, and

source records between a UNL directive and a LIST directive are not listed. The difference between two printed source record numbers indicates how many source records are not listed.

The next field contains the Location Counter value as a hexadecimal value. In the example, >0156 is the Location Counter value. Not all directives affect the Location Counter. If the directive does not affect the Location Counter, the field is left blank. Of the directives that the Assembler lists, the IDT, REF, DEF, DXOP, EQU, SREF, LOAD, COPY, and END directives leave the Location Counter field blank.

The third field, >C820 in the example, contains the hexadecimal representation of the object code placed in the location by the Assembler. The apostrophe (') following the third field of the second line in the example indicates that the contents, >012B, are program-relocatable. A quote (") in this location indicates that the location is data-relocatable, while a plus sign (+) indicates that the label is relocatable with respect to a common segment. All machine instructions and the BYTE, DATA, and TEXT directives use this field for object code. The EQU directive places the value corresponding to the label in the object code field.

The fourth field contains the first 60 characters of source statement as you wrote it. Spacing in this field is determined by the spacing in the source statement. Thus, source statement fields are aligned in the listing only when they are aligned in the same character positions in the source statements.

The machine instruction used in the example specifies the symbolic memory addressing mode for both operands. This causes the instruction to occupy three words of memory and three lines of the listing. The object code corresponds to the operands in the order in which they appear in the source statement.

15.1.1 Error Messages

The Assembler processes fatal and nonfatal error messages. Fatal errors stop the assembly process with the appropriate error message displayed on the screen as shown.

Fatal Errors

SYMBOL TABLE OVERFLOW
CANT GET COMMON
CANT GET MEMORY
DSR ERROR XXXX

The XXXX field following DSR ERROR contains the first two bytes of the Peripheral Access Block which contain the error code. See Section 18.2 for more information.

Nonfatal errors do not stop the assembly process. Instead, an error message is printed following the statement containing the error and is also displayed on the screen. Each error gives the type of error and the number of the statement in which it occurred.

Nonfatal Errors

***** SYNTAX ERROR - nnnn
***** INVALID REF - nnnn
***** OUT OF RANGE - nnnn
***** MULTIPLE SYMBOLS - nnnn
***** INVALID MNEMONIC - nnnn
***** BAD FWD REFERENCE - nnnn
***** INVALID TERM - nnnn
***** INVALID REGISTER - nnnn
***** SYMBOL TRUNCATION - nnnn
***** UNDEFINED SYMBOL - nnnn
***** COM TABLE OVERFLOW - nnnn
***** PEND ASSUMED - nnnn
***** DEND ASSUMED - nnnn
***** CEND ASSUMED - nnnn
***** END ASSUMED - nnnn
***** COPY ERROR - nnnn

If there are any undefined symbols in the assembly, the undefined symbols are listed at the end of the listing under the heading THE FOLLOWING SYMBOLS ARE UNDEFINED.

15.2 OBJECT CODE

The Assembler produces object code that may be loaded directly into the computer. If it is not compressed, object code consists of records containing up to 71 ASCII characters each. This format permits correction of minor errors by using the Editor to edit the object code. The file format is DISPLAY, FIXED 80.

15.2.1 Object Code Format

The object code consists of variable-sized records. Each record consists of a number of character tags, each followed by up to two fields. Most fields are numeric and consist of four hexadecimal digits. The length of the character fields is described in the following sections on each of the tag characters.

When the Assembler has no more data or the record is full, it writes the tag character 7, followed by the checksum field and the tag character F, which requires no fields. The Assembler then fills the rest of the record with blanks and a sequence number and then begins a new record with the appropriate tag character. The last record of an object module has a colon (:) in the first character position of the record, followed by the identification code 99/4 AS.

The tag characters used by the Assembler are 0, 1, 2, 3, 4, 5, 6, 7, 9, A, B, C, and F. You may substitute a tag character 8 for a tag character 7 in order to have the Loader ignore the checksum. Tag character I is ignored. Any other tag produces an error.

Tag character 0 is used for program identification. Field 1 contains the number of bytes of program-relocatable code, and field 2 contains an eight-character program identifier assigned to the program by an IDT directive. When you do not include an IDT directive, the second field contains blanks. The Loader uses the program identifier to identify the program and the number of bytes of program-relocatable code to determine the load bias for the next module or program. The Assembler places a single tag character 0 at the beginning of each program.

Tag characters 1 and 2 are employed with entry addresses. Tag character 1 is used when the entry address is absolute. Tag character 2 is applicable when the entry address is relocatable. The field contains the entry address. One of these tags may appear at the end of the object code file. The Loader uses the field value to determine the entry point at which execution starts when the loading is complete.

Tag characters 3 and 4 are used for external references. Tag character 3 is used when the last appearance of the symbol in Field 2 of the tag is in program-relocatable code. Tag character 4 is employed when the last appearance of the symbol is not in relocatable code. Field 1 contains the location of the last appearance of the symbol. The six-character symbol in Field 2 is the external reference. For each external reference in a program, the object code contains a tag character with a location or an absolute zero and the symbol that is referenced. When Field 1 of the tag character contains absolute zero, no location in the program requires the address that corresponds to the reference. When Field 1 of the tag character contains a location, the address corresponding to the reference is placed in the location specified and the location's previous value is used to point to the next location unless the value is an absolute zero.

Tag characters 5 and 6 are used for external definitions. Tag character 5 is applicable when the location is program-relocatable. Tag character 6 is used when the location is absolute. Field 1 provides the link to the external definition, and field 2 contains the six-character symbol of the external definition.

Tag character 7 precedes the checksum, which is an error-detection word. The checksum is the two's complement of the sum of the 8-bit ASCII values of the characters in the record from the first tag of the record through the checksum tag.

Tag character 8 may be inserted in place of tag character 7 so that the Loader ignores the checksum. The field contains the previous checksum.

Tag characters 9 and A are used with load addresses for following data. Tag character 9 is used when the load address is absolute, while tag character A indicates that the load address is program-relocatable. The field contains the address at which the following data word is to be loaded. A load address is required for a data word that is to be placed in memory at some address other than the next address.

Tag characters B and C are used with data words. Tag character B is used when the data is absolute, such as an instruction word or a word that contains text characters or absolute constants. Tag character C is used for a word that contains a program-relocatable address. The field contains the data word. The Loader places the data word in the memory location specified in the preceding load address field or in the memory location that follows the preceding data word.

Tag character F indicates the end of the record.

The following table summarizes the character tags.

Character Tag Summary

Tag	Use	Field 1	Field 2
0	Program Identification	Program Length	Program ID
1	Entry Point Definition	Absolute Address	
2	Entry Point Definition	Relocatable Address	
3	External References	Relocatable Address of Chain	Symbol
4	External References	Absolute Address of Chain	Symbol
5	External Definitions	Relocatable Address	Symbol
6	External Definitions	Absolute Address	Symbol
7	Checksum Indicator	Checksum	
8	Checksum Ignore	Any Value	
9	Load Address	Absolute Value	
A	Load Address	Relocatable Address	
B	Data	Absolute Value	
C	Data	Relocatable Address	
F	End of Record		

15.2.2 Compressed Object Code Format

This format is a condensed version of normal object code. Compressed object code results in a considerable savings of diskette space compared to the normal object format. You cannot change or edit compressed object code. Instead, change the source code and reassemble it.

15.3 CHANGING OBJECT CODE

Correction of the object code that the Assembler produces may only require changing a character or a word. You can edit the object code using the Editor. Because the changes may cause a checksum error when the checksum is verified as the record is loaded, you must change the 7 tag character to an 8.

You can only change uncompressed object code. After you have finished editing, save the file in fixed 80 format.

For best results, when more extensive changes are required, change the source code and reassemble rather than writing additional object code records.

15.4 MACHINE LANGUAGE FORMAT

Some of the data words preceded by tag character B represent machine instructions. Comparing the source listing with the object code fields identifies the data words that represent machine instructions. The following table shows the manner in which the bits of the machine instructions relate to the operands in the source statements for each format of machine instruction.

Machine Instruction Formats

Format	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
I	1	1	X	WB	Td		D				Ts		S			
I	1	0	X	WB	Td		D				Ts		S			
I	0	1	X	WB	Td		D				Ts		S			
II	0	0	0	1	X	X	X	X	DISP							
III, IX	0	0	1	X	X	X	D				Ts		S			
IV	0	0	1	1	0	X	NUM				Ts		S			
V	0	0	0	0	X	0	X	X	COUNT				REG			
VI	0	0	0	0	0	1	X	X	X	X	Ts		S			
VII	0	0	0	0	0	0	1	1	X	X	X	0	0	0	0	0
VIII	0	0	0	0	0	0	1	0	X	X	X	0	REG			
X	0	0	0	0	0	0	1	1	0	0	1		REG			

Key:

X	A bit of the operation code that is either 0 or 1, according to the specific instruction in the format.
WB	A bit of the operation code that is 0 in instructions that operate on words and 1 in instructions that operate on bytes.
Td	A pair of bits that specify the addressing mode of the destination operand. 00 specifies Workspace Register addressing. 01 specifies Workspace Register indirect addressing. 10 specifies symbolic memory addressing when D equals 0 and indexed memory addressing when D is not equal to 0. 11 specifies Workspace Register indirect auto-increment addressing.
D	The Workspace Register for the destination operand.
Ts	A pair of bits that specify the addressing mode of the source operand as shown for Td.
S	The Workspace Register for the source operand.
NUM	The number of bits to be transferred.
DISP	A two's complement number that represents a displacement.
REG	A Workspace Register address.
COUNT	A shift count.

15.5 OUTPUT EXAMPLE

The example of a crash sound, given in Section 20.4.3, produces the listing and object code shown below.

15.5.1 Listing

The following is the listing produced when the crash program is assembled.

```
 99/4 ASSEMBLER
VERSION 1.1                                                    PAGE 0001
  0001                  *
  0002                  *  Example Program to make a crash sound.
  0003                  *
  0004                        REF  VMBW
  0005                        DEF  CRASH
  0006                  *
  0007       1000       BUFFER EQU  >1000   VDP RAM buffer used by sound generator.
  0008                  *
  0009 0000    01       H01    BYTE >01
  0010                        EVEN
  0011                  *
  0012                  CRASH
  0013 0002 0200              LI   R0,BUFFER   Load VDP RAM buffer address.
       0004 1000
  0014 0006 0201              LI   R1,CDATA    Pointer to the sound data.
       0008 0038'
  0015 000A 0202              LI   R2,32   32 bytes to move to the VDP RAM buffer.
       000C 0020
  0016 000E 0420              BLWP @VMBW   Move to VDP RAM buffer.
       0010 0000
  0017                  *
  0018                  LOOP
  0019 0012 0300              LIMI 0   Disable VDP interrupt.
       0014 0000
  0020 0016 020A              LI   R10,BUFFER   Load sound table address.
       0018 1000
  0021 001A C80A              MOV  R10,@>83CC   Load pointer to the table.
       001C 83CC
  0022 001E F820              SOCB @H01,@>83FD   Set VDP RAM flag.
```

ASSEMBLER OUTPUT

```
      0020 0000'
      0022 83FD
 0023 0024 D820         MOVB @H01,@>83CE   Start sound processing.
      0026 0000'
      0028 83CE
 0024 002A 0300         LIMI 2   Enable VDP interrupt.
      002C 0002
 0025              *
 0026              LOOP2
 0027 002E D820         MOVB @>83CE,@>83CE   Check if time is up.
      0030 83CE
      0032 83CE
 0028 0034 13EE         JEQ  LOOP   Repeat the sound.
 0029 0036 10FB         JMP  LOOP2   Wait until finished.
 0030              *
 0031 0038   03   CDATA  BYTE >03,>9F,>E4,>F2,5
      0039   9F
      003A   E4
      003B   F2
      003C   05
 0032 003D   02         BYTE >02,>E4,>F0,12
      003E   E4
      003F   F0
      0040   0C
 0033 0041   02         BYTE >02,>E4,>F2,10
      0042   E4
      0043   F2
      0044   0A
 0034 0045   02         BYTE >02,>E4,>F4,8
      0046   E4
```



```
      0047   F4
      0048   08
 0035 0049   02         BYTE >02,>E4,>F6,6
      004A   E4
      004B   F6
      004C   06
 0036 004D   02         BYTE >02,>E4,>F8,4
```

```
       004E   E4
       004F   F8
       0050   04
 0037  0051   02          BYTE >02,>E4,>FA,2
       0052   E4
       0053   FA
       0054   02
 0038  0055   01          BYTE >01,>FF,0
       0056   FF
       0057   00
 0039                     END


 99/4 ASSEMBLER
VERSION 1.1                                                    PAGE 0003
     BUFFER  1000     ' CDATA  0038     D CRASH  0002     ' H01    0000
     LOOP    0012     ' LOOP2  002E       R0     0000       R1     0001
     R10     000A       R11    000B       R12    000C       R13    000D
     R14     000E       R15    000F       R2     0002       R3     0003
     R4      0004       R5     0005       R6     0006       R7     0007
     R8      0008       R9     0009     E VMBW   0010
  0000 ERRORS
```

15.5.2 Object Code

The following is the object code produced when the crash program is assembled.

```
00058      0000B0100B0200B1000B0201C0038B0202B0020B0420B00007F39AF      0001
A0012B0300B0000B020AB1000BC80AB83CCBF820C0000B83FDBD820C00007F2C6F      0002
A0028B83CEB0300B0002BD820B83CEB83CEB13EEB10FBB039FBE4F2B05027F23BF      0003
A003EBE4F0B0C02BE4F2B0A02BE4F4B0802BE4F6B0602BE4F8B0402BE4FA7F21BF      0004
A0054B0201BFF007FC8CF                                                  0005
30010VMBW  50002CRASH 7FAD1F                                           0006
:       99/4 AS                                                        0007
```

SECTION 16: UTILITIES AND PREDEFINED SYMBOLS

Several utilities are provided to give you simple access to many of the resources of the TI Home Computer. With these utilities, you can change the display, access the Device Service Routine for peripheral devices such as disk drives and printers, scan the keyboard, link your program to GPL routines that perform a variety of useful tasks, and link to the Editor/Assembler Loader. This section discusses these utilities and many of the predefined symbols. Other predefined symbols, used for sound and speech, are discussed in Sections 20 and 22.

Normally, it is difficult to write to and read from VDP RAM and VDP Registers because they, like GROM and speech devices, are memory mapped. To read from most memory-mapped devices, you must first write a value to a specific address, wait while the data is obtained, and then read the data from another address. To write to most memory-mapped devices, a similar process occurs: put the data in an address, write a value to an address to signify that the data is to be written, and then wait while the data is written. This requires detailed knowledge of the addresses to use and how to use them.

The utilities and predefined symbols are loaded at the same time as the Loader. You can make them available by mentioning them in a REF statement at the beginning of your assembly language program, accessing them with a BLWP instruction, and using Registers to pass arguments. They are also loaded into the Memory Expansion unit for use when the TI BASIC subroutines INIT or LOAD are called.

The utilities are predefined in the REF/DEF table at memory locations >3F38 through >3FFF. They use UTILWS, starting at address >2094, as utility Workspace Registers. All parameters are passed through your program's Workspace Registers. The USRWSP area at >20BA may be used for your Workspace Registers.

The following list gives each of the utilities predefined in the REF/DEF table and describes briefly what each does. Sections 16.1 and 16.2 provide a more detailed discussion.

Name	Use
VSBW	Writes a single byte to VDP RAM.
VMBW	Writes multiple bytes to VDP RAM.
VSBR	Reads a single byte from VDP RAM.
VMBR	Reads multiple bytes from VDP RAM.
VWTR	Writes a single byte to a VDP Register.
KSCAN	Scans the keyboard.
GPLLNK	Links your program to Graphics Programming Language routines.
XMLLNK	Links your program to the assembly language routines in the console ROM or in RAM.
DSRLNK	Links your program to Device Service Routines.
LOADER	Links your program to the Loader to load TMS9900 tagged object code.

Several general use addresses are predefined with symbols. You may use them instead of the addresses they represent so that you do not have to memorize the addresses. Their use is described in Section 16.3.

Name	Address	Data Contained
SCAN	>000E	Entry address of the keyboard scan utility.
UTLTAB	>2022	Utility table entry address.
PAD	>8300	The scratch pad used by TI BASIC, GPL, TI BASIC, and other programs. You may use some areas. See Section 24.3.1 for a detailed description of this area.
GPLWS	>83E0	GPL Workspace.

Some addresses that are useful for accessing memory-mapped devices are predefined with symbols. You may use these symbols in your own memory access routines instead of the utilities described above. The use of these symbols is described in Sections 16.4 and 16.5.

Name	Address	Data Contained
VDPWA	>8C02	VDP RAM write address.
VDPRD	>8800	VDP RAM read data.
VDPWD	>8C00	VDP RAM write data.
VDPSTA	>8802	VDP RAM status.
GRMWA	>9C02	GROM/GRAM write address.
GRMRA	>9802	GROM/GRAM read address.
GRMRD	>9800	GROM/GRAM read data.
GRMWD	>9C00	GROM/GRAM write data.

16.1 VDP RAM ACCESS UTILITIES

Several utilities provide access to Video Display Processor RAM. All parameters are passed through your program's Workspace Registers. The utilities are described below.

Note: Before you change VDP Register 1, put a copy of it at address >83D4. The bit that turns the screen on and off, which is used when no key is pressed for a certain time, is in VDP Register 1 and is stored at that address. Therefore, if you do not put the copy there, the screen returns to a prior state when you press a key.

BLWP @VSBW: VDP RAM Single Byte Write--Writes the value in the most-significant byte of Register 1 to the VDP RAM address indicated in Register 0.

Register 0: Address in VDP RAM.

Register 1: Most-significant byte contains the value to be written.

For example, if Register 0 is >1000 and Register 1 is >2345, the instruction sets address >1000 in VDP RAM equal to >23.

BLWP @VMBW: VDP RAM Multiple Byte Write--Writes the number of bytes indicated in Register 2 from the RAM buffer pointed to by Register 1 to the VDP RAM buffer pointed to by Register 0. The RAM buffer is normally included in your program space with the BSS instruction.

Register 0: Starting address of the buffer in VDP RAM.

Register 1: Starting address of the buffer in RAM.

Register 2: The number of bytes to be written.

For example, if Register 0 is >1000, Register 1 is >2345, and Register 2 is >0014, the instruction sets addresses >1000 through >1013 in VDP RAM equal to the 20 bytes starting at address >2345 in the Memory Expansion unit.

BLWP @VSBR: VDP RAM Single Byte Read--Reads a byte from the VDP RAM

address indicated in Register 0 and places it in the most-significant byte of Register 1.

Register 0: Address in VDP RAM.

Register 1: Value is placed in the most-significant byte.

For example, if Register 0 is >1000 and address >1000 contains >12, the instruction sets the most-significant byte of Register 1 to >12.

BLWP @VMBR: VDP RAM Multiple Byte Read--Reads the number of bytes indicated in Register 2 from the RAM buffer pointed to by Register 0 and places them in the CPU RAM buffer pointed to by Register 1.

Register 0: Starting address of the buffer in VDP RAM.

Register 1: Starting address of the buffer in RAM.

Register 2: The number of bytes to be read.

For example, if Register 0 is >1000, Register 1 is >2345, and Register 2 is >0014, the instruction reads the 20 bytes starting at address >1000 in VDP RAM and sets addresses >2345 through >2358 in CPU RAM equal to those bytes.

BLWP @VWTR: VDP RAM Write Register--Writes the value in the least-significant byte of Register 0 to the VDP Register indicated in the most-significant byte of Register 0.

Register 0: Least-significant byte contains the value to be written. Most-significant byte indicates the VDP Register to be written to.

For example, if Register 0 is >010E, the instruction loads VDP Register 1 with the value >0E. See Section 16.4 for more information on the VDP Registers.

16.2 EXTENDED UTILITIES

Five utilities, called extended utilities, allow you to access the routines built into the Home Computer. KSCAN allows you to use the routine that scans the keyboard. GPLLNK allows you to link to Graphics Programming Language routines. XMLLNK allows you to use routines in ROMs. DSRLNK allows you to link to Device Service Routines. LOADER allows you to load assembly language programs.

16.2.1 KSCAN

The KSCAN utility allows you to access the key scan routine in the computer. To use this utility, you must include REF KSCAN in your program, select the keyboard device to be checked, and call the utility with the instruction

```
BLWP    @KSCAN
```

Then check the STATUS byte to see if a key has been pressed for the first time and check an address to see what the key was. You can also check to see if the Wired Remote Controllers have been moved.

Select the keyboard device to be checked by placing a byte at address >8374. A value of >00 checks the entire keyboard. A value of >01 checks the left side of the keyboard and places the values from Wired Remote Controller unit number one in addresses >8376 (Y-position) and >8377 (X-position). A value of >02 checks the right side of the keyboard and places the values from Wired Remote Controller unit number two in addresses >8376 and >8377. The values that may be returned in the addresses for the Wired Remote Controller are >04 (up or right), >00 (center), and >FC (down or left). For more information on key units, see the explanation of the CALL KEY subprogram in the User's Reference Guide.

The status bit which indicates if a key has been pressed can be tested with a compare ones corresponding (COC) instruction. The STATUS byte is at address >837C and is as shown below.

bit	0	1	2	3	4	5	6	7
	H	GT	COND	CARRY	OVF	0	0	0

When KSCAN is called, bit 2 of the STATUS byte is set if a key was pressed that was different from the key pressed on the last call to KSCAN. The ASCII value of the key pressed is placed at address >8375. If no key was pressed, this address contains >FF.

For example, if your program places >00 at address >8374, BLWP @KSCAN is called, and the B key is pressed for the first time, then bit 2 of the byte at address >83C7 is set, the value >42 is in address >8375.

16.2.2 GPLLNK

GPLLNK allows you to use routines written in Graphics Programming Language. These routines allow you to perform such tasks as loading character sets, giving tones, allocating space for strings, and so on. Some of the routines are described below.

To use GPLLNK, you must include REF GPLLNK in your assembly language program, set the STATUS byte (at address >837C) to >00, branch to GPLLNK with BLWP, and provide the address of the routine in the console as data. For example, the following instructions branch to the routine which loads the standard character set.

```
REF     GPLLNK
.
.
.
CLR     R1
MOVB    R1,@>837C
BLWP    @GPLLNK
DATA    >0016
.
.
.
```

The GPL routines described below return to your program when they finish executing. This utility lets you access any address in GROM. However, other routines than the ones given may branch to other code or have other side effects. To be sure that a routine returns to your program, check to see that it ends with a GPL return instruction (>00). You must also check that the routine does not use memory areas that your program is also using.

In the routines shown below, FAC (the Floating Point Accumulator) starts at address >834A and ARG (which contains arguments) starts at address >835C. The STATUS byte is at address >837C. VSPTR is at address >836E.

16.2.2.1 General Purpose GPL Routines

The following are some useful general purpose GPL routines which you may access with GPLLNK.

DATA >0016 Load Standard Character Set--Loads the standard character set into VDP RAM.

Input: FAC is a pointer to the beginning address in VDP RAM where characters are to be loaded.

DATA >0018 Load Small Capitals Character Set--Loads the small capitals character set into VDP RAM.

Input: FAC is a pointer to the beginning address in VDP RAM where characters are to be loaded.

DATA >0020 Execute Power-Up Routine--Initializes the system.

Output: The sound and VDP circuits are cleared and the default values for the VDP Registers, character set, color table, and status block are loaded. The available VDP RAM size is stored at >8370.

DATA >0034 Accept Tone--Issues the tone associated with correct input.

DATA >0036 Bad Response Tone--Issues the tone associated with incorrect input.

DATA >0038 Get String Space Routine--Allocates a memory space in VDP RAM with a specified number of bytes. This routine should not be used outside the TI BASIC environment. If there is not enough space, the routine does a "garbage collection" to eliminate temporary strings and then tries again. If there is still not enough space, the routine issues the MEMORY FULL error message.

Input: Addresses >830C and >830D should contain the number of bytes to be allocated.

Output: Address >831C points to the allocated string space and address >831A points to the first free address in VDP RAM. The four bytes at addresses >8356 through >8359 are used by this routine. The FAC area may be destroyed if a garbage collection is done.

Note: Although this routine is designed to allocate a string space in VDP RAM, it is also useful for assigning space for the Peripheral Access Block (PAB) and data buffer required by a DSR. See Section 18.2 for a description of Peripheral Access Blocks.

DATA >003B Bit Reversal Routine--Provides a mirror image of a byte. This routine is used to form a mirror image of a character definition.

Input: FAC is the address of the data in VDP RAM. FAC+2 (>834C) is the number of bytes to be reversed.

Output: In each byte, bits 0 and 7, 1 and 6, 2 and 5, and 3 and 4 are exchanged. CPU RAM addresses >8300 through >8340 are destroyed.

DATA >003D Cassette DSR Routine--Accesses the cassette DSR routine.

Input: The Peripheral Access Block and data buffer must be set up in VDP RAM prior to the call. The screen offset for TI BASIC is >60 and >00 outside the TI BASIC environment. The screen start address must be >00 for the prompts issued by the cassette DSR. FAC is the device name (for example, "CS1"). Address >8356 points to the first character after the name in the PAB. Addresses >8354 and >8355 are the length of the name (for example, >0003 for "CS1"). The word at address >83D0 should be set to >0000. Address >836D must be set to >08 to indicate a DSR call. The STATUS byte must be >00.

Output: The cassette DSR prompts for the operation of the cassette.

DATA >004A Load Lower-Case Character Set (TI-99/4A only)--Loads the lower-case character set into VDP RAM.

Input: FAC is a pointer to the beginning address in VDP RAM where characters are to be loaded.

16.2.2.2 Mathematical Routines

When errors occur during the execution of these floating point routines, they are indicated at address >8354 with the error codes listed.

Code	Error Description
01	Overflow
02	Syntax
03	Integer overflow on conversion
04	Square root of a negative number
05	Negative number to non-integer power
06	Logarithm of a non-positive number
07	Invalid argument in a trigonometric function

The abbreviations for these routines given in parentheses are the TI BASIC functions which call the routines.

DATA >0014 Convert Number to String (STR)--Converts a floating point number to an ASCII string.

Input: FAC contains the eight bytes defining the number. FAC+11 (>8355), if set to 0, indicates that the output string is to be in TI BASIC format. Otherwise, the output is in FIX mode. If fix mode is indicated, then FAC+12 (>8356) and FAC+13 (>8357) must contain data. FAC+12 is the number of significant digits. It contains 0 to express overflow from the calculation range. FAC+13 indicates the number of digits to the right of the decimal point. A negative value disables the FIX mode.

Output: FAC is modified, FAC+11 (>8355) contains the least significant byte of the address where the result string is located. >8300 must be added to FAC+11 to find the address of the result string. FAC+12 (>8356) contains the length of the result string in bytes.

DATA >0022 Greatest Integer Function (INT)--Computes the greatest integer contained in the value.

Input: FAC contains the floating point value.

Output: FAC contains the result. For positive numbers, the integer is the truncated value. For negative numbers, the integer is the truncated value plus one. The STATUS byte is affected.

DATA >0024 Involution Routine--Raises a number to a specified power.

Input: FAC is the exponent value. ARG is the base value.

Output: FAC is the result in floating point format. The result is computed as EXP(exponent-value*LOG[ABS(base-value)]). The STATUS byte is affected. Locations >8375 and >8376 are destroyed and the word content of VSPTR is decremented by eight.

DATA >0026 Square Root Routine (SQR)--Computes the square root of a number.

Input: FAC is the input value.

Output: FAC is the square root of the input value. The STATUS byte is affected. Addresses >8375 and >8376 are destroyed.

DATA >0028 Exponent Routine (EXP)--Computes the inverse natural logarithm of a number.

Input: FAC is the input value.

Output: FAC is the result value. The STATUS byte is affected. Addresses >8375 and >8376 are destroyed.

DATA >002A Natural Logarithm Routine (LOG)--Computes the natural logarithm of a number.

Input: FAC is the input value.

Output: FAC is the result value. The STATUS byte is affected. Addresses >8375 and >8376 are destroyed.

DATA >002C Cosine Routine (COS)--Computes the cosine of a number expressed in radians.

Input: FAC is the input value.

Output: FAC is the result value. The STATUS byte is affected. Addresses >8375 and >8376 are destroyed.

DATA >002E Sine Routine (SIN)--Computes the sine of a number expressed in radians.

Input: FAC is the input value.

Output: FAC is the result value. The STATUS byte is affected. Addresses >8375 and >8376 are destroyed.

DATA >0030 Tangent Routine (TAN)--Computes the tangent of a number expressed in radians.

Input: FAC is the input value.

Output: FAC is the result value. The STATUS byte is affected. Addresses >8375 and >8376 are destroyed.

DATA >0032 Arctangent Routine (ATN)--Computes the arctangent of a number expressed in radians.

Input: FAC is the input value.

Output: FAC is the result value. The STATUS byte is affected. Addresses >8375 and >8376 are destroyed.

Note: A GPL routine is executed in the main program of the Editor/Assembler. Prior to transfer to the Editor/Assembler, the GPLLNK routine sets a flag. However, the Editor/Assembler main program checks the GPL status bit for error handling before it checks for the flag set by the GPLLNK routine. Thus, you must reset that status bit at address >837C before calling GPLLNK. Otherwise, a meaningless error message is returned.

16.2.3 XMLLNK

The XMLLNK utility allows you to link an assembly language program to a routine in ROM or to branch to a routine in the Memory Expansion unit. The ROM routines perform such tasks as floating point arithmetic, stack arithmetic, string to number conversions, and so on.

In linking to console ROM routines, you refer to table entries with DATA instructions. These table entries contain the addresses of the routines which you wish to execute. The following describes the routines contained in the tables and the addresses of the tables. Each entry in the table takes up two bytes, so the addresses accessed are incremented by two for each entry.

Table Number	Function	Address
>0	Floating Point Routines	Specified at >0CFA
>1	Conversion and TI BASIC routines	Specified at >0CFC
>2	Memory Expansion unit	>2000
>3	TI BASIC enhancement	>3FC0
>4	TI BASIC enhancement	>3FE0
>5	Peripheral ROM	>4010
>6	Peripheral ROM	>4030
>7	ROM in Command Module	>6010

To link to a routine in ROM, the DATA instruction is followed by a word that specifies the table and entry you wish to use. The first byte of the word indicates the table and entry you are referring to, with the second byte equal to >00.

The first nybble of the byte is from >0 through >7, indicating the table which you wish to use. The second nybble of the byte is from >0 through >F. When doubled, it indicates the offset from the beginning of the table, from >00 through >1E. For

example, DATA >1100 indicates the first table, two bytes from the beginning (the address of the beginning of the table is stored at address >0CFA); DATA >3400 indicates the third table, eight bytes from the beginning, or address >3FC8; and DATA >5C00 indicates the fifth table, >18 bytes from the beginning, or address >4028.

When you branch to XMLLNK, the address of the routine to be executed is obtained from the table, and then the routine is executed starting at the address obtained. For example, the following sequence branches to the convert integer to floating point routine at address >2006 in the Memory Expansion unit.

```
REF     XMLLNK
.
.
.
BLWP    @XMLLNK
DATA    >2300
```

You may also use the XMLLNK utility to branch to an address and start executing there. In this case, the DATA instruction is the address to which you wish to branch. The first bit of the word must be on, so the address may be from >8000 through >FFFF.

This use of the XMLLNK has the same effect as using the BL instruction (see Section 7.2), except that the GPL Workspace Registers are used during the instructions executed after the branch. When the program returns, then your Workspace Registers are again used.

For example, the following sequence branches to address >C13A in the Memory Expansion unit.

```
REF     XMLLNK
.
.
.
BLWP    @XMLLNK
DATA    >C13A
```

16.2.3.1 ROM Routines

The following describe some of the routines in ROM that you can access with XMLLNK, giving the data required, the input, and the output. Since XMLLNK accesses routines in the console, care must be taken to obtain the intended result. You must be sure that the GPL Workspace Registers are not changed, the memory space used in the routine is set up properly, and the utility returns to the calling program on completion.

FAC (the Floating Point Accumulator) starts at address >834A, ARG (which contains arguments) starts at address >835C, and VSPTR is at address >836E. The STATUS byte is at address >837C. All overflow errors, except in Convert Floating Point to Integer (CFI), return >01 at address >8354.

DATA >0600 Floating Point Addition (FADD)--Adds two values.

Input: FAC is the first value and ARG is the second value.

Output: FAC is the result of the addition.

DATA >0700 Floating Point Subtraction (FSUB)--Subtracts two values.

Input: FAC is the value to be subtracted. ARG is the value from which FAC is subtracted.

Output: FAC is the result of the subtraction.

DATA >0800 Floating Point Multiplication (FMUL)--Multiplies two values.

Input: FAC is the multiplier. ARG is the multiplicand.

Output: FAC is the result of the multiplication.

DATA >0900 Floating Point Division (FDIV)--Divides two values.

Input: FAC is the divisor. ARG is the dividend.

Output: FAC is the result of the division.

DATA >0A00 Floating Point Compare (FCOM)--Compares two floating point numbers.

Input: ARG is the first argument. FAC is the second argument.

Output: The STATUS byte is affected. The high bit is set if ARG is logically higher than FAC. The greater than bit is set if ARG is arithmetically greater than FAC. The equal bit is set if ARG and FAC are equal.

DATA >0B00 Value Stack Addition (SADD)--Adds using a stack in VDP RAM.

Input: VSPTR contains the address in VDP RAM where the left-hand term is located. FAC is the right-hand value.

Output: FAC is the result of the addition.

DATA >0C00 Value Stack Subtraction (SSUB)--Subtracts using a stack in VDP RAM.

Input: VSPTR contains the address in VDP RAM where the left-hand term is located. FAC is the value to be subtracted.

Output: FAC is the result of the subtraction.

DATA >0D00 Value Stack Multiplication (SMUL)--Multiplies using a stack in VDP RAM.

Input: VSPTR contains the address in VDP RAM where the multiplicand is located. FAC is the multiplier value.

Output: FAC is the result of the multiplication.

DATA >0E00 Value Stack Division (SDIV)--Divides using a stack in VDP RAM.

Input: VSPTR contains the address in VDP RAM where the dividend is located. FAC is the divisor value.

Output: FAC is the result of the division.

DATA >0A00 Floating Point Compare (FCOM)--Compares two floating point numbers.

Input: ARG is the first argument. FAC is the second argument.

Output: The STATUS byte is affected. The high bit is set if ARG is logically higher than FAC. The greater than bit is set if ARG is arithmetically greater than FAC. The equal bit is set if ARG and FAC are equal.

DATA >0B00 Value Stack Addition (SADD)--Adds using a stack in VDP RAM.

Input: VSPTR contains the address in VDP RAM where the left-hand term is located. FAC is the right-hand value.

Output: FAC is the result of the addition.

DATA >0C00 Value Stack Subtraction (SSUB)--Subtracts using a stack in VDP RAM.

Input: VSPTR contains the address in VDP RAM where the left-hand term is located. FAC is the value to be subtracted.

Output: FAC is the result of the subtraction.

DATA >0D00 Value Stack Multiplication (SMUL)--Multiplies using a stack in VDP RAM.

Input: VSPTR contains the address in VDP RAM where the multiplicand is located. FAC is the multiplier value.

Output: FAC is the result of the multiplication.

DATA >0E00 Value Stack Division (SDIV)--Divides using a stack in VDP RAM.

Input: VSPTR contains the address in VDP RAM where the dividend is located. FAC is the divisor value.

Output: FAC is the result of the division.

DATA >0F00 Value Stack Compare (SCOMP)--Compares a value in the VDP RAM stack to the value in FAC.

Input: VSPTR contains the address in VDP RAM where the value to be compared is located. FAC is the other value in the comparison.

Output: The STATUS byte is affected. The high bit is set if the value pointed to by VSPTR is logically higher than FAC. The greater than bit is set if the value pointed to by VSPTR is arithmetically greater than FAC. The equal bit is set if the value pointed to by VSPTR and FAC are equal.

DATA >1000 Convert String to Number (CSN)--Converts an ASCII string to a floating point number.

Input: FAC+12 is the address of the string in VDP RAM.

Output: FAC is the result of the conversion in floating point format.

DATA >1200 Convert Floating Point to Integer (CFI)--Converts a floating point number to an integer.

Input: FAC is the floating point number to be converted.

Output: FAC is the one-word integer value. The maximum value is >FFFF. If an overflow occurs, FAC+10 (>8354) is set to the overflow error code, >03.

DATA >2300 Convert Integer to Floating Point (CIF)--Converts an integer to a floating point number.

Input: FAC is the one word integer value to be converted.

Output: FAC is the floating point result.

Note: This routine is loaded in the Memory Expansion unit by the Editor/Assembler Loader. It is not loaded by the TI Extended BASIC Loader and may not be used in TI Extended BASIC.

16.2.4 DSRLNK

DSRLNK links an assembly language program to any Device Service Routine (DSR) or subprogram in ROM. The data given is >8 for linkage to a Device Service Routine and >10 for linkage to a subprogram. Before this routine is called, a Peripheral Access Block (PAB) must be set up in VDP RAM. A PAB is a block of memory that contains information about the file to be accessed. In addition, CPU RAM addresses >8356 through >8357 must contain a pointer to the device or subprogram name length in the PAB. See Section 18.2 for information on building a PAB.

After the routine is executed, information is passed back to your assembly language program in the UTLTAB area (see Section 16.3.2). For instance, suppose that the following sequence of instructions is executed.

```
REF     DSRLNK
.
.
.
BLWP    @DSRLNK
DATA    >8
```

If no errors occur, the equal bit in the Status Register is reset on return from DSRLNK. If an I/O error occurs, the equal bit is set, and the error code is stored in the most-significant byte of Register 0 of the calling program's Workspace.

Note: This routine does not work for cassette access because it only searches ROM DSRs and the cassette DSR is in GPL in the GROM. To access the cassette, BLWP @GPLLNK with DATA >3D must be used. See Section 16.2.2.1 for more information on accessing cassettes.

16.2.5 LOADER

LOADER loads TMS9900 tagged object code such as the Assembler produces. (See Section 19 for a description of the Loader and Section 15 for a description of tagged object code.) This utility is only used when you wish to load a file from your assembly language program. Otherwise, selecting LOAD AND RUN from the Editor/Assembler or executing a CALL LOAD statement from TI BASIC is recommended.

Before this utility is called, a Peripheral Access Block (PAB) set for OPEN mode must be set up in VDP RAM. A PAB is a block of memory that contains information about the file to be accessed. See Section 18.2 for information on building a PAB. In addition, CPU RAM addresses >8356 through >8357 must contain a pointer to the device or file name length in the PAB.

For example, the following loads a file if the proper PAB has been set up.

```
REF     LOADER
.
.
.
BLWP    @LOADER
```

After the utility is executed, information is passed back to your assembly language program in the UTLTAB area (see Section 16.3.2). To access this data, you must include the instruction

```
REF     UTLTAB
```

in your program.

Address	Name	Information
>2022	UTLTAB	Entry address specified by a 1 or 2 tag.
UTLTAB+>2	FSTHI	First free address in high memory.
UTLTAB+>4	LSTHI	Last free address in high memory.
UTLTAB+>6	FSTLOW	First free address in low memory.
UTLTAB+>8	LSTLOW	Last free address in low memory.

If no errors occur, the equal bit in the Status Register is reset on return from LOADER. If an I/O or load error occurs, the equal bit is set and the error code is stored in the most-significant byte of Register 0 of the calling program's Workspace.

Note: *The Loader does not close the file when errors occur, so in that case you must call the DSR to close the file.*

16.3 PREDEFINED SYMBOLS

Several predefined symbols which can be used in place of addresses are included in the REF/DEF table when the utilities are loaded. You must list the symbols in a REF statement in your assembly language program to use them.

16.3.1 SCAN

SCAN is the entry address of the keyboard scan routine in the console and is set to >000E. With SCAN, you have direct access to the keyboard scan routine KSCAN. However, before using this address you must supply information to the GPL Workspace Registers and load the Registers before branching to this routine. KSCAN does this for you. For example, the following instructions branch to this address.

```
REF     SCAN,GPLWS
.
.
.
LWPI    GPLWS
BL      @SCAN
LWPI    MYWS
```

16.3.2 UTLTAB

UTLTAB is the utility table. It contains information provided by DSRLNK (see Section 16.2.4) and LOADER (see Section 16.2.5). The following table lists the elements of UTLTAB, their standard addresses, the names often used to refer to them, and the information that they contain.

Reference	Address	Name	Information
UTLTAB	>2022	UTLTAB	Entry address.
UTLTAB+>2	>2024	FSTHI	First free address in high memory.
UTLTAB+>4	>2026	LSTHI	Last free address in high memory.
UTLTAB+>6	>2028	FSTLOW	First free address in low memory.
UTLTAB+>8	>202A	LSTLOW	Last free address in low memory.
UTLTAB+>A	>202C	CHKSAV	Checksum.
UTLTAB+>C	>202E	FLGPTR	Pointer to the flag in the PAB.
UTLTAB+>E	>2030	SVGPRT	GPL return address.
UTLTAB+>10	>2032	SAVCRU	CRU address of the peripheral.
UTLTAB+>12	>2034	SAVENT	Entry address of the DSR or subprogram.
UTLTAB+>14	>2036	SAVLEN	Device or subprogram name length.
UTLTAB+>16	>2038	SAVPAB	Pointer to the device or subprogram name in the PAB.
UTLTAB+>18	>203A	SAVVER	Version number of the DSR.

16.3.3 PAD

PAD is the address of the beginning of the scratch pad in RAM. It is from addresses >8300 through >83FF. It is used by assembly language programs, console routines, and Graphics Programming Language routines. See the Appendix for a complete description of this memory area.

16.4 VDP ACCESS

You can access VDP RAM and VDP Registers directly instead of using the utilities described at the beginning of this section. The following sections demonstrate how to use predefined addresses to access various memory areas.

In accessing memory in VDP, allow enough time for the completion of the read from or write to memory. This can most easily be accomplished by following the instruction that uses the address with a NOP or SWPB instruction. Also, most of these addresses require that you read or write the least-significant byte first.

16.4.1 VDPWA

VDPWA is the address of the VDP RAM Write Address Register and is set to >8C02. This Register must be prepared when you wish to access VDP RAM. To set the address you plan to access in VDP RAM, move the two-byte address into this location. The least-significant byte is transferred first, followed by a delay (with NOP or SWPB), and then the most-significant byte is transferred. If data is to be written, the most-significant two bits of the address must be 01.

For example, if the address to which you plan to read is in Register 1, the following code loads that address.

```
REF     VDPWA           Refers to the address.
.
.
.
SWPB    R1              Gets the least-significant byte first.
MOVB    R1,@VDPWA       Writes the least-significant byte.
SWPB    R1              Takes time and gets the most-significant byte.
MOVB    R1,@VDPWA       Writes the most-significant byte.
```

The VDP RAM Register write address is auto-incrementing, so you can write successive bytes without modifying the write address.

You may also use this utility to change VDP Registers by setting the first bit, specifying the VDP Register in the second nybble, and giving the value to write in the second byte. For example, the following code changes VDP Register 2 to >01.

```
REF     VDPWA           Refers to the address.
.
.
.
LI      R1,>8201
SWPB    R1              Gets the least-significant byte first.
MOVB    R1,@VDPWA       Writes the least-significant byte.
SWPB    R1              Takes time and gets the most-significant byte.
MOVB    R1,@VDPWA       Writes the most-significant byte.
```

16.4.2 VDPRD

VDPRD is the address of the VDP RAM Read Data Register and is set to >8800. The address of the VDP RAM must be set as described in Section 16.4.1. Data can then be read from VDPRD.

For example, if Register 1 contains the address of VDP RAM that you plan to read, the following code puts the value from that address into Register 0.

```
REF     VDPWA,VDPRD     Refers to the addresses.
.
.
.
SWPB    R1              Gets the least-significant byte first.
MOVB    R1,@VDPWA       Writes the least-significant byte.
SWPB    R1              Takes time and gets the most-significant byte.
MOVB    R1,@VDPWA       Writes the most-significant byte.
NOP                     Takes time.
MOVB    @VDPRD,R0       Reads the data into Register 0.
```

The VDP RAM Register read address is auto-incrementing, so you can read successive bytes without modifying the read address.

16.4.3 VDPWD

VDPWD is the address of the VDP RAM Write Data Register and is set to >8C00. The address in VDP RAM to which you plan to move data must first be specified with VDPWA with the most-significant two bits set to 01. Data is then moved to the VDP RAM write data address when the instruction MOVB @VDPWD is executed.

For example, if Register 1 contains the address to which you plan to write and the most-significant byte of Register 3 contains the data you wish to write, the following statements move the value from Register 3 to the specified address.

```
REF     VDPWA,VDPWD     Refers to the addresses.
.
.
.
LI      R2,>4000        Sets Register 2 with the two most-significant
                        bits equal to 01.
SWPB    R1              Gets the least-significant byte first.
MOVB    R1,@VDPWA       Writes the least-significant byte.
SWPB    R1              Takes time and gets the most-significant byte.
SOC     R2,R1           Sets the two most-significant bits in Register 1
                        to 01.
MOVB    R1,@VDPWA       Writes the most-significant byte.
NOP                     Takes time.
MOVB    R3,@VDPWD       Writes the data from the most-significant byte
                        of Register 3.
```

The VDP RAM write address is auto-incrementing, so you can write successive bytes without modifying the write address.

16.4.4 VDPSTA

VDPSTA is the address of the VDP RAM Read Status Register and is set to >8802. The Status Register is read by moving it from that address to the desired destination. The status is the most significant byte of the address.

For example, the following code moves the status byte to Register 1.

```
REF     VDPSTA          Refers to the address.
.
.
.
MOVB    @VDPSTA,R1      Reads the Status Register.
```

The VDP Status Register contains the following information.

Bit	Information
0	VDP interrupt flag. Set if a VDP interrupt has occurred. This flag may be read even if interrupts have been disabled with the LIMI 0 instruction. The flag is cleared by reading the Status Register or by resetting VDP.
1	Five sprites flag. Set if there are five or more sprites on a screen line. The flag is cleared by reading the Status Register or by resetting VDP.
2	Coincidence flag. Set if two or more sprites have overlapping pixels, including sprites that are transparent and sprites that are off the bottom of the screen. The flag is cleared by reading the Status Register or by resetting VDP.
3-7	Fifth sprite number. Equal to the number of the fifth sprite on a line if the Coincidence flag is set. The value is cleared by reading the Status Register or by resetting VDP.

16.5 GROM ACCESS

You can look at information in the GROM or GRAM in the Command Module by using the four addresses described below.

In accessing memory in the GROM, allow enough time for the completion of the read from or write to memory. This can most easily be accomplished by following the instruction that uses the address with a NOP or SWPB instruction. Also, most of these addresses require that you read or write the most-significant byte first.

16.5.1 GRMWA

GRMWA is the address of the GROM Write Address Register and is set to >9C02. This Register must be prepared when a GROM address is to be accessed. To set the address to be accessed in the GROM, move the two-byte address into this location. The most-significant byte is transferred first, followed by a delay (with NOP or SWPB), and then the second byte is transferred.

For example, if the address to which you plan to read or write is in Register 3, the following code loads that address.

```
REF     GRMWA           Refers to the address.
.
.
.
MOVB    R3,@GRMWA       Writes the most-significant byte.
SWPB    R3              Takes time.
MOVB    R3,@GRMWA       Writes the least-significant byte.
SWPB    R3              Return Register to its original state.
```

16.5.2 GRMRA

GRMRA is the address of the GROM Read Address Register and is set to >9802. The address is read by moving the two-byte address from the read address memory location to the destination. The most-significant byte is transferred first, followed by the least-significant byte. The address must be decremented by one. After being read, the GROM address value is indeterminate and must be restored or reset before further data access can occur.

For example, the following statements move the address into VALUE and VALUE+1.

```
REF     GRMRA              Refers to the address.
.
.
.
MOVB    @GRMRA,@VALUE      Reads first byte of the address.
NOP                        Takes time.
MOVB    @GRMRA,@VALUE+1    Reads second byte of the address.
DEC     @VALUE             Corrects the address.
```

16.5.3 GRMRD

GRMRD is the address of the GROM Read Data Register and is set to >9800. The address of the GROM must be set as described in Section 16.5.1. Data can then be read from GRMRD.

For example, if Register 3 contains the address of GROM that you plan to read, the following statements put the value from that address into Register 1.

```
REF     GRMWA,GRMRD        Refers to the addresses.
.
.
.
MOVB    R3,@GRMWA          Writes the most-significant byte.
SWPB    R3                 Takes time.
MOVB    R3,@GRMWA          Writes the least-significant byte.
SWPB    R3                 Takes time.
MOVB    @GRMRD,R1          Reads the data into Register 1.
```

16.5.4 GRMWD

GRMWD is the address of the GROM Write Data Register and is set to >9C00. No data can be written to a GROM. However, if a GRAM is in place, you may write data.

For example, if Register 1 contains the address to which you plan to write and Register 3 contains the data you wish to write, the following statements move the value from Register 3 to the specified address.

```
REF     GRMWA,GRMWD     Refers to the addresses.
.
.
.
MOVB    R1,@GRMWA       Writes the most-significant byte.
SWPB    R1              Takes time and gets the least-significant by
MOVB    R1,@GRMWA       Writes the least-significant byte.
NOP                     Takes time.
MOV     R3,@GRMWD       Writes the data from Register 3.
```

SECTION 17: TI BASIC SUPPORT

The Editor/Assembler Command Module contains seven TI BASIC subprograms that can be used in addition to the subprograms described in the User's Reference Guide. They are described below and discussed in greater detail in Section 17.1.

Name	Use
INIT	Loads utilities and tables into the Memory Expansion unit and clears any previously loaded programs.
LOAD	Loads an assembly language file or pokes values into CPU RAM.
LINK	Passes control from TI BASIC to an assembly language program.
PEEK	Reads bytes from CPU RAM into TI BASIC variables.
PEEKV	Reads bytes from VDP RAM into TI BASIC variables.
POKEV	Pokes values into VDP RAM.
CHARPAT	Returns the value of character patterns.

Examples of using some of these subprograms are given in Section 17.1.8.

In addition, the Editor/Assembler diskette labeled Part A contains the file BSCSUP which contains several TI BASIC support utilities. These utilities allow you to access variables and values passed in the parameter list of the TI BASIC subprogram LINK. In addition, ERR allows you to return an error to the calling TI BASIC program.

These utilities use their own Workspace Registers. The Workspace Registers starting at USRWS are loaded by the Name Search Routine in the utility before branching to your program and are available for your use unless your program runs because an entry point was specified after the END directive (see Section 14.5.2). In this case, you must specify that Workspace yourself or provide your own Workspace Registers. All parameters are passed through the calling program's Workspace Registers. An example of using some of these utilities is given in Section 17.2.6.

The following list gives the available utilities and describes briefly what each does. They are described in greater detail in Section 17.2.

Name	Use
NUMASG	Makes a numeric assignment.
STRASG	Makes a string assignment.
NUMREF	Gets a numeric parameter.
STRREF	Gets a string parameter.
ERR	Reports errors.

17.1 INTERFACE WITH TI BASIC

Seven subprograms, which are included in the Editor/Assembler Command Module, are added to TI BASIC for use in interfacing with assembly language programs. The LOAD, POKEV, PEEK, and PEEKV subprograms can be used whether or not the Memory Expansion unit is attached. They are described in the following sections.

17.1.1 CALL INIT

The format of the INIT subprogram is

```
CALL INIT
```

with no parameters. The INIT subprogram tests to be sure that the Memory Expansion unit is properly connected, loads utility routines from the Editor/Assembler module into the Memory Expansion unit starting at address >2000, and loads REF/DEF tables in the Memory Expansion unit at addresses >3F38 through >3FFF.

The INIT subprogram should be called before assembly language programs are loaded by TI BASIC. If the INIT subprogram is called while an assembly language program is in memory, it removes all information relating to that program, making the program inaccessible. However, the program itself may remain in memory.

17.1.2 CALL LOAD

The format of the LOAD subprogram depends on the use to which it is put. It may be used to load an object file such as is produced by the Assembler or to "poke" data directly into memory locations.

17.1.2.1 Loading a Program with LOAD

If you use the LOAD subprogram to load an assembly language object file, the format is

```
CALL LOAD("object-filename"["object-filename",...])
```

The object filename is a string expression such as DSK1.OBJFILE. The file must contain assembly language object code, such as that produced by the Assembler. More than one file can be loaded at a time by separating the files you want to load with commas.

For example, the statement

```
CALL LOAD("DSK1.OBJ1","DSK1.OBJ2")
```

loads the files OBJ1 and OBJ2 from the diskette in Disk Drive 1.

Relocatable code is loaded starting at the first available address, which is set to >A000 by the INIT subprogram (described in Section 17.1.1). Room is reserved for the program according to the length specified in the character tag 0 field in the object file. (See Section 15.2 for a description of character tags.) Absolute code is loaded as specified in the assembly language program.

CAUTION

You must take extreme care that absolute code is really needed and works properly. Loading data into memory already being used by TI BASIC can cause the system to stop functioning so that you must turn the computer off and back on in order to continue.

If more than one program is loaded, the additional programs are loaded in the memory following the previous program. See Section 19 for more information on the Loader.

17.1.2.2 Poking Data with LOAD

If you use the LOAD subprogram to put data directly into memory ("poking"), the format is

```
CALL LOAD(address,value[,value,...[,"",address,value[,value,...]]])
```

The address is a numerical expression or variable from -32768 through 32767. Addresses from 0 through 32767 represent >0000 through >7FFF. Addresses from -32768 through -1 represent >8000 through >FFFF expressed in two's-complement form. To access an address above 32767, subtract 65536 from it. The values, which can be repeated, are decimal numbers which specify the byte values to be loaded starting at the address specified. For example, the statement

```
CALL LOAD(-16384,255,21)
```

places the values >FF and >15 in the bytes starting at address >C000.

You can specify a new address and the values to be loaded starting at that address by separating the last value from the new address with an empty string (""). For example, the statement

```
CALL LOAD(-16384,255,21,"",8192,85)
```

loads the same data in the same addresses as the previous program and also loads the value >55 at address >2000.

You can use the LOAD subprogram to load a program directly into memory. However, you must enter the program name in the REF/DEF table so that the program can be run with the LINK subprogram (described in Section 17.1.3).

To enter the program name in the REF/DEF table, use the PEEK subprogram (described in Section 17.1.4) to find the values at addresses >2028 and >202A. These addresses contain the First Free Address in Low memory (FFAL) and Last Free Address in Low memory (LFAL), respectively. These values must differ by at least eight bytes to have space for your program name and address. Change LFAL to a value eight less than its old value, and then load the program name, up to six bytes, starting at the new LFAL address, followed by two bytes which give the starting address of the program. For example, suppose LFAL is >3F38, your program name is OBJ1, and it starts at address >8300. Change LFAL to >3F30 and load OBJ1, two spaces, and >83 00 into addresses >3F30 through >3F37 with the statement

```
CALL LOAD(16176,79,66,74,49,32,32,131,00)
```

17.1.3 CALL LINK

The format of the LINK subprogram is

CALL LINK("program-name"[,parameter-list])

The program-name is from one through six characters that give the name of the assembly language program as it appears in the REF/DEF table. The assembly language program must be in memory, and its name must be in the REF/DEF table. See the explanation of LOAD (Section 17.1.2) for more information.

The optional parameter-list contains parameters you wish to pass from the TI BASIC program to your assembly language program. Using parameters is discussed in greater detail in Section 17.1.3.1.

For example, the statement

CALL LINK("START",1,3)

links the TI BASIC program to the assembly language program START, with the values 1 and 3 passed to it.

The following actions occur when the CALL LINK statement is executed.

1. The utility program checks to see that the assembly language program name is from one through six characters long. If the name is of the right length, the Name Link Routine (which is part of the utility program) looks up the name of the program called in the REF/DEF table, starting at the lowest address. The Name Link Routine then pushes the program name on the value stack.

   The Loader gives an error if you have duplicate names in DEF instructions, and loading stops.

2. When the program name has been located in the REF/DEF table, the Name Link Routine branches to the program with a direct assembly language branch instruction. In order to return to TI BASIC, your assembly language program must retain the values in Workspace Registers 11, 13, 14, and 15 and restore those values before ending.

3. Your assembly language program is executed.

4. When your program has finished executing, the utility branches to an error routine if an error has been detected. Otherwise, it clears the stack and returns to the TI BASIC program normally. Address >8310 contains the value stack pointer in use by the TI BASIC interpreter.

17.1.3.1 Passing Arguments with LINK

You can pass up to 16 arguments from your TI BASIC program to your assembly language program. If a simple variable (any variable except an expression) is passed, any changes made in the value of that variable in your assembly language program also change the value on return to your TI BASIC program. Entire arrays are passed by following them with parentheses. If the array contains more than one dimension, the dimensions beyond the first are indicated by placing commas between the parentheses.

For example, the following are all simple variables whose values can be changed. The last one is a two-dimensional array.

```
A,B$,VAR(3),Q$(,)
```

If you wish to pass the value of a simple variable, but do not need the assembly language program to make changes in it, surround it with parentheses. However, arrays cannot be passed by value. For example, you can pass all but the last of the variables listed above without having their values affected on return to your TI BASIC program by listing them as follows.

```
(A),(B$),(VAR(3))
```

In addition, constants and expressions, such as A+3, do not have their values changed on return to your TI BASIC program.

The arguments are passed to the assembly language program through an identifier list in CPU RAM. Address >8312 contains the number of arguments in the parameter list. The argument identifier, which specifies the type of argument, is located at addresses >200A through >2019. Each identifier is one byte in length. The values are 0 for a numeric expression, 1 for a string expression, 2 for a numeric variable, 3 for a string variable, 4 for a numeric array, and 5 for a string array.

Note: You do not need to know exacly how arguments are passed if you use the utilities described in Section 17.2

More information on each argument is stored in an eight-byte value stack in VDP RAM.

- If the argument is a numeric expression, the identifier is 0 and the stack contains the value of the numeric expression in radix 100 notation. In radix 100 notation, a number is from 1.000000000000 through 99.999999999999 multiplied by 100 raised to a power from -64 to 64. The first byte in the value stack indicates the exponent of the value of the numeric expression. If the exponent is positive, the byte value is 64 more than the exponent. If the exponent is negative, the byte value is obtained by subtracting the exponent from 64. For example, if the exponent is 2, the byte is 66 or >42. If the exponent is -3, the byte is 61 or >3D. If the number is negative, the first word (the exponent byte and the first byte of the value) is given in two's-complement form.

  The remaining seven bytes indicate the value of the number. To find these seven bytes, the number, in decimal form but with the decimal point missing, is converted to hexadecimal notation. For example, the following show how several values are expressed in radix 100 form.

Decimal Value	Radix 100 Notation	Stack Value
7	7×100^0	>40 >07 >00 >00 >00 >00 >00 >00
70	70×100^0	>40 >46 >00 >00 >00 >00 >00 >00
2,345,600	2.3456×100^3	>43 >02 >22 >38 >00 >00 >00 >00
23,456,000	23.456×100^3	>43 >17 >2D >3C >00 >00 >00 >00
0	0×100^0	>00 >00 >XX >XX >XX >XX >XX >XX
-7	-7×100^0	>BF >F9 >00 >00 >00 >00 >00 >00
-70	-70×100^0	>BF >BA >00 >00 >00 >00 >00 >00
-2,345,600	-2.3456×100^3	>BC >FE >22 >38 >00 >00 >00 >00

Note: The value 0 is expressed by >00 in each of the first two bytes and undefined values in the remaining six bytes.

- If the argument is a string expression, the identifier is 1. Bytes 0 and 1 contain >001C, byte 2 contains >65 (the string tag used by the TI BASIC interpreter), byte 3 is not used, bytes 4 and 5 contain a pointer to the value of the string in VPD RAM, and bytes 6 and 7 contain the length of the string. Byte 6 is always zero because the maximum string length is 255 characters.

- If the argument is a numeric variable or a numeric array element, the identifier list contains a 2. Bytes 0 and 1 contain a pointer to the variable's symbol table entry in VDP RAM, byte 2 contains zero, byte 3 is not used, and bytes 4 and 5 contain a pointer to the eight-byte value of the variable in VDP RAM.

- If the argument is a string variable or a string array element, the identifier list contains a 3. Bytes 0 and 1 contain a pointer to the variable's symbol table entry in VDP RAM, byte 2 contains >65 (the string tag used by the TI BASIC interpreter), byte 3 is not used, bytes 4 and 5 contain a pointer to the string's value in VDP RAM, and bytes 6 and 7 contain the string length.

- If the argument is a numeric array of the form A() A(,), and so on, the identifier list contains a 4. Bytes 0 and 1 contain a pointer to the array's symbol table entry in VDP RAM, byte 2 contains zero, byte 3 is not used, and bytes 4 and 5 contain a pointer to the array's value space in VDP RAM. The value space has two bytes for each dimension, indicating the maximum index for that dimension. Following the dimension information are the values, stored in radix 100 notation. Note that in a numeric array the array elements are stored in consecutive eight-byte sections of memory.

- If the argument is a string array, it is similar to the entry for a numeric array except the identifier list contains a 5 and byte 2 of the stack entry contains >65. Thus bytes 0 and 1 contain a pointer to the array's symbol table entry in VDP RAM, byte 2 contains >65, byte 3 is not used, and bytes 4 and 5 contain a pointer to the array's value space in VDP RAM. The value space for a string array contains two bytes for each dimension, indicating the maximum index. Following the dimension information are two bytes for each array element, which are used as a pointer to the element's value string in VDP RAM. Note that in a numeric array the array elements are stored in consecutive eight-byte sections of memory, while in a string array the elements are not usually in order.

17.1.4 CALL PEEK

The format of the PEEK subprogram is

CALL PEEK(address,variable-list[,"",...])

The address is a numerical expression or variable from -32768 through 32767. Addresses from 0 through 32767 represent >0000 through >7FFF. Addresses from -32768 through -1 represent >8000 through >FFFF expressed in two's-complement form. To access an address above 32767, subtract 65536 from it.

The PEEK subprogram reads bytes from CPU RAM and, starting at the address, assigns those values to the numeric variables in the variable-list. You can read values starting at more than one address by separating the last value in the variable-list from the next address with an empty string (""). For example, the statement

CALL PEEK(8192,A,B,C(8),"",-24576,X)

places the value from address 8192 (>2000) in A, the value from address 8193 (>2001) in B, the value from address 8194 (>2002) in C(8), and the value from address -24576 (>A000) in X.

17.1.5 CALL PEEKV

The format of the PEEKV subprogam is

CALL PEEKV(address,variable-list[,"",...])

The address is a numerical expression or variable from 0 through 16383, corresponding to VDP RAM addresses >0000 through >3FFF.

The PEEKV subprogram reads bytes from VDP RAM and, starting at the address, assigns those values to the numeric variables in the variable-list. You can read values starting at more than one address by separating the last value in the variable-list from the next address with an empty string (""). For example, the statement

CALL PEEKV(784,A,B,C(8),"",2,X)

places the value from address 784 (>0310) in A, the value from address 785 (>0311) in B, the value from address 786 (>0312) in C(8), and the value from address 2 (>0002) in X.

Note: Using an address higher than 16383 (>3FFF) can cause the system to stop functioning so that you must turn the computer off and back on in order to continue.

17.1.6 CALL POKEV

The format of the POKEV subprogram is

CALL POKEV(address,value-list[,"",...])

The address is a numerical expression or variable from 0 through 16383, corresponding to VDP RAM addresses >0000 through >3FFF.

The POKEV subprogram writes bytes to VDP RAM from the value-list starting at the address. You can write values starting at more than one address by separating the last value in the value-list from the next address with an empty string (""). For example, the statement

CALL POKEV(784,30,30,30,"",2,V)

places the value 30 (>1E) in addresses 784 (>0310), 785 (>0311), and 786 (>0312), and places the value of V in address 2 (>0002).

Note: Using an address higher than 16383 (>3FFF) can cause the system to stop functioning so that you must turn the computer off and back on in order to continue.

17.1.7 CALL CHARPAT

The format of the CHARPAT subprogram is

CALL CHARPAT(character-code,string-variable[,...])

The character-code is any character number from 32 to 159. The 16-character hexadecimal pattern identifier associated with the character code is returned in the string-variable. The pattern identifiers for characters 32 through 95 are normally reserved for ASCII characters and are initially defined by TI BASIC. They can be

changed, and characters 96 through 159 defined, by the CHAR subprogram. See the User's Reference Guide for more information.

17.1.8 TI BASIC Examples

The following program initializes memory, loads the file SPRITE from the diskette in Disk Drive 1, and executes the program, starting at the entry point BEGIN.

```
100 CALL INIT
110 CALL LOAD("DSK1.SPRITE")
120 CALL LINK("BEGIN")
```

The program below initializes memory, loads the TI BASIC support utilities (BSCSUP) and the file DSK1.TEST, and executes the program, starting at the entry point TEST. The parameters passed to the assembly language program are the numeric array A and the string expression HELLO. After the assembly language program has finished running, the program prints the value of A(9).

```
100 DIM A(30)
110 CALL INIT
120 CALL LOAD("DSK1.BSCSUP","DSK1.TEST")
130 CALL LINK("TEST",A(),"HELLO")
140 PRINT A(9)
```

The following commands read the one-word value at CPU RAM address >8370 and calculate and print the value. The value contains the highest memory location available in VDP RAM.

```
CALL PEEK(-31888,A,B)
VALUE=A*256+B
PRINT VALUE
```

The program below loads color table 16 at VDP RAM address >0310. As the program executes, the background color of the space characters on the screen changes rapidly.

```
100 FOR I=1 TO 16
110 CALL POKEV(784,16+I)
120 NEXT I
```

17.2 TI BASIC SUPPORT UTILITIES

The TI BASIC support utilities are contained in the file BSCSUP on the Editor/Assembler diskette labeled Part A. These utilities help you find the values, and assign values to the variables, passed in the parameter-list of the LINK subprogram.

The five utilities are NUMASG, STRASG, NUMREF, STRREF, and ERR. They are in relocatable code and are about 900 bytes long. You can use them in your assembly language program by listing them in a REF statement. To load them, put the statement

```
CALL LOAD("DSK1.BSCSUP")
```

in your TI BASIC program. The Loader loads them in the Memory Expansion unit and puts their names in the REF/DEF table.

An example of the use of these utilities is given in Section 17.2.6.

17.2.1 Numeric Assignment--NUMASG

This utility lets you assign a value to a numeric variable passed as an argument in the TI BASIC subprogram LINK.

For assignments to a simple numeric variable, place 0 in Workspace Register 0. For an assignment to an array, place the array element number in Workspace Register 0. With OPTION BASE 0 (the default from TI BASIC) in effect, the element number ranges from 0 to the maximum number of elements minus 1. With OPTION BASE 1 (from TI BASIC) in effect, the element number ranges from 1 to the maximum number of elements. See the User's Reference Guide for information on OPTION BASE. The element number for multiple dimension arrays is found by counting through the first elements, then the second elements, and so on. For instance, if an array has been defined as A(5,5,5) and the base is 0, array element A(1,2,3) is given in Workspace Register 0 as $1 * 6^2 + 2 * 6 + 3$ or 51.

Place the argument number, as a full word, in Workspace Register 1. The argument number gives the order of appearance of the variable in the parameter-list of the LINK subprogram. For example, if the LINK subprogram statement is

```
CALL LINK("START",A,B)
```

A is argument number 1 and B is argument number 2.

The floating point variable is assigned in the Floating Point Accumulator at address >834A. Numbers in the Floating Point Accumulator are kept in radix 100 notation. See Section 17.1.3.1 for an explanation of radix 100 notation.

The utility is accessed by BLWP @NUMASG. For example, suppose the TI BASIC statement CALL LINK("PROG",A,B,C) was executed to pass control to the assembly language program. Then if the Floating Point Accumulator, starting at address >834A, contains >41 23 45 00 00 00 00 00, Register 0 is >00, and Register 1 is >03, then BLWP @NUMASG assigns 356.9 to C.

As a further example, the following program segment assigns the value 3 to the third argument passed by the LINK subprogram.

```
        REF     NUMASG
        .
        .
        .
FAC     EQU     >834A
NUMBER  DATA    >4003,>0000,>0000,>0000
        .
        .
        .
        LI      R4,4
        LI      R3,NUMBER
        LI      R2,FAC
LOOP    MOV     *R3+,*R2+   Load floating point number 03 into FAC area.
        DEC     R4
        JNE     LOOP
        CLR     R0
        LI      R1,3        Third numeric variable in list.
        BLWP    @NUMASG
        .
        .
        .
```

17.2.2 String Assignment--STRASG

This utility allows a string to be assigned to a string variable passed as an argument in the TI BASIC subprogram LINK. The utility allocates space for the string in VDP RAM, copies the string into VDP RAM, and assigns the string to the selected variable. It then modifies the original argument stack entry to point to the new string.

Before using this utility, your program must create the string in the Memory Expansion unit with the first byte in the string giving the length of the string. Then the utility is called with the string address in Register 2 and the argument number in Register 1 as a full word. The number must be the same as it appeared in the CALL LINK statement. The utility is accessed by BLWP @STRASG.

For assignments to a simple string variable, place 0 in Workspace Register 0. For an assignment to an array, place the array element number in Workspace Register 0. With OPTION BASE 0 (the default from TI BASIC) in effect, the element number ranges from 0 to the maximum number of elements minus 1. With OPTION BASE 1 (from TI BASIC) in effect, the element number ranges from 1 to the maximum number of elements. See the User's Reference Guide for information on *OPTION* BASE. The element number for multiple dimension arrays is found by counting through the first elements, then the second elements, and so on. For instance, if an array has been defined as A$(5,5,5) and the base is 0, then array element A$(1,2,3) is given in Workspace Register 0 as $1 * 6^2 + 2 * 6 + 3$ or 51.

For example, if your program has placed >00 in Register 0, >02 in Register 1, >C000 in Register 2, and >02 48 49 in the addresses starting at >C000 (*where >02 is the* number of characters in the string, >48 is the ASCII code for H, and >49 is the ASCII code for I), and the TI BASIC statement CALL LINK("START",A,B$,C) is executed, then BWLP @STRASG sets B$ equal to HI.

17.2.3 Get Numeric Parameter--NUMREF

This utility allows you to get the value of a numeric parameter specified in the TI BASIC subprogram LINK. If the parameter is an array, Register 0 contains the element number. Otherwise, Register 0 contains 0. Register 1 contains the parameter number. The value of the numeric parameter is returned in the Floating Point Accumulator area starting at address >834A in radix 100 form. For details, see Section 17.2.1 on numeric assignment. The utility is accessed by BLWP @NUMREF.

17.2.4 Get String Parameter--STRREF

This utility allows you to get the value of a string parameter specified in the TI BASIC subprogram LINK. If the parameter is an array, Register 0 contains the element number. Otherwise, Register 0 contains 0. Register 1 contains the parameter number. Register 2 contains the starting address of the string in the Memory Expansion unit. Put the length of the buffer into which you are going to read the string in the first byte of the starting address. If the string length actually read exceeds the number specified, an error is issued. Otherwise the actual length is placed in the first byte. For details, see Section 17.2.2 on string assignment. The utility is accessed by BLWP @STRREF.

17.2.5 Error Reporting--ERR

This utility transfers control to the error reporting routine in the TI BASIC interpreter. The assembly language program can report any existing TI BASIC error or warning upon return to TI BASIC. Upon return, Workspace Register 0 contains the error code in the most significant byte. The utility is accessed by BLWP @ERR. In order to obtain a meaninful error code, the Peripheral Access Block address must be stored at address >831C prior to the ERR call.

The error messages that can be issued from your program are given on the next page.

Error Code	Message
>00	I/O error (bad name).
>01	I/O error (write protected).
>02	I/O error (bad attribute).
>03	I/O error (illegal operation).
>04	I/O error (buffer full).
>05	I/O error (read past EOF).
>06	I/O error (device error).
>07	I/O error (file error).
>08	Memory full (closes files).
>09	Not applicable; signifies an incorrect statement.
>0A	Bad tag.
>0B	Checksum error.
>0C	Duplicate definition.
>0D	Unresolved references.
>0E	Not applicable; signifies an incorrect statement.
>0F	Program not found.
>10	Incorrect Statement.
>11	Bad name.
>12	Can't continue.
>13	Bad value.
>14	Number too big.
>15	String-number mismatch.
>16	Bad argument.
>17	Bad subscript.
>18	Name conflict.
>19	Can't do that.
>1A	Bad line number.
>1B	For-next error.
>1C	I/O error.
>1D	File error.
>1E	Input error.
>1F	Data error.
>20	Line too long.
>21	Memory full (does not close files).
>22 - >FF	Unknown error.

Note that on any error code >00 through >07, an I/O ERROR message is displayed on the screen. The ERR utility then transfers control to the TI BASIC error-handling routine, which reads the I/O error code from the Peripheral Access Block.

17.2.6 TI BASIC Utilities Example

The following program demonstrates how TI BASIC support utilities are called from an assembly language program. The TI BASIC program file loads and runs an assembly language program called DSK1.STRINGO. Together, these programs assign B$="HAPPY BIRTHDAY" to an array element A$(X) and display A$(X) on the screen. The user specifies X when the program is executing.

The TI BASIC program loads the BSCSUP file and the file STRINGO and links to the program called STRING. TI BASIC lists the arguments X, B$, and A$() to be passed in the CALL LINK statement. Before transferring control to the assembly language program, it prompts for the element number X.

The assembly language program STRING reads the element number X, reads the string B$, and assigns B$ to the array element A$(X). Since Workspace Register 11 was not altered by this program, the RT instruction can be used to return to TI BASIC.

The TI BASIC program then displays A$(X), which is HAPPY BIRTHDAY, on the screen.

```
100 DIM A$(15)
110 B$="HAPPY BIRTHDAY"
120 CALL INIT
130 CALL LOAD("DSK1.BSCSUP","DSK2.TRY")
140 INPUT "ELEMENT NUMBER?":X
150 IF X>15 THEN 140
160 CALL LINK("STRING",X,B$,A$())
170 PRINT A$(X)
180 END
```

```
         DEF   STRING
         REF   STRREF,STRASG,NUMREF
*
FAC      EQU   >834A
*
BUFFER   BYTE  >1F
         BSS   >1F
*
STRING
         CLR   R0
         LI    R1,1          Read numeric value X.
         BLWP  @NUMREF
*
         MOV   @FAC,R5
         ANDI  R5,>00FF      Keep the element number.
*
         CLR   R0
         LI    R1,2          Second argument B$.
         LI    R2,BUFFER     Points to the buffer area.
         BLWP  @STRREF       Read in the string value.
*
         MOV   R5,R0         Element number.
         LI    R1,3          Assign the string to A$(X).
         LI    R2,BUFFER
         BLWP  @STRASG
*
         RT
         END
```

SECTION 18: FILE MANAGEMENT

With assembly language, you can control the way in which files are accessed. This section describes the file management system as it is provided. By making appropriate changes, you can construct your own system to interface with devices in other ways.

The file management system regards all devices (except the screen and keyboard) as identical. Different Device Service Routines (DSRs) are used for different devices, but they all appear the same to the assembly language programmer. They support both random access and sequential files, and files with records of both fixed and variable length. The following sections describe the way in which you access the DSRs.

18.1 FILE CHARACTERISTICS

A file consists of a collection of data groupings called logical records. These records do not necessarily correspond with the the physical divisions of the data in the file. For example, a logical record often does not correspond to a sector on a diskette. File input and output (I/O) are done on a logical record basis. Manipulation of physical records is handled by the DSR.

The records on sequential files can only be read from, or written to, in sequential order. This is appropriate for printers, modems, cassettes, and some kinds of data files. The records on sequential files can be of either fixed or variable length.

The records on relative files can be read from, or written to, in either sequential order or in random order. You can only use relative files on diskettes. The records on relative files are of fixed length.

Each record on a file has a number from zero up to one less than the number of records in the file. You use these record numbers to specify which record to access on relative files.

When a file is created, its characteristics must be defined. Most of these characteristics cannot be changed later in the file's existence. The characteristics of files are discussed below.

18.1.1 File Type--DISPLAY or INTERNAL

The file type attribute specifies the format of the data in the file.

- DISPLAY sets the file type to contain displayable or printable character strings. Each data record corresponds to one print line.
- INTERNAL sets the file type to contain data in internal machine format.

The file type attribute is not significant to the DSR. It is merely passed on without affecting the actual data stored.

18.1.2 Mode of Operation--INPUT, OUTPUT, UPDATE, or APPEND

A file is opened for a specific mode of operation.

- INPUT specifies that the contents of the file can be read from but not written to.
- OUTPUT specifies that the file is being created. Its contents can be written to but not read from.
- UPDATE specifies that the contents of the file can be both written to and read from.
- APPEND specifies that data can be added to the end of the file but data cannot be read.

The DSR determines whether a specific mode for an I/O operation can be accepted by the given device. For example, the TI Thermal Printer can only be opened in OUTPUT mode.

18.2 PERIPHERAL ACCESS BLOCK (PAB) DEFINITION

DSRs are accessed through a Peripheral Access Block (PAB). The format of the PAB is the same for every peripheral. In a program that you write, the only difference between peripherals is that some of them do not allow every option provided for in the PAB. An example of using a PAB is given in Section 18.3.

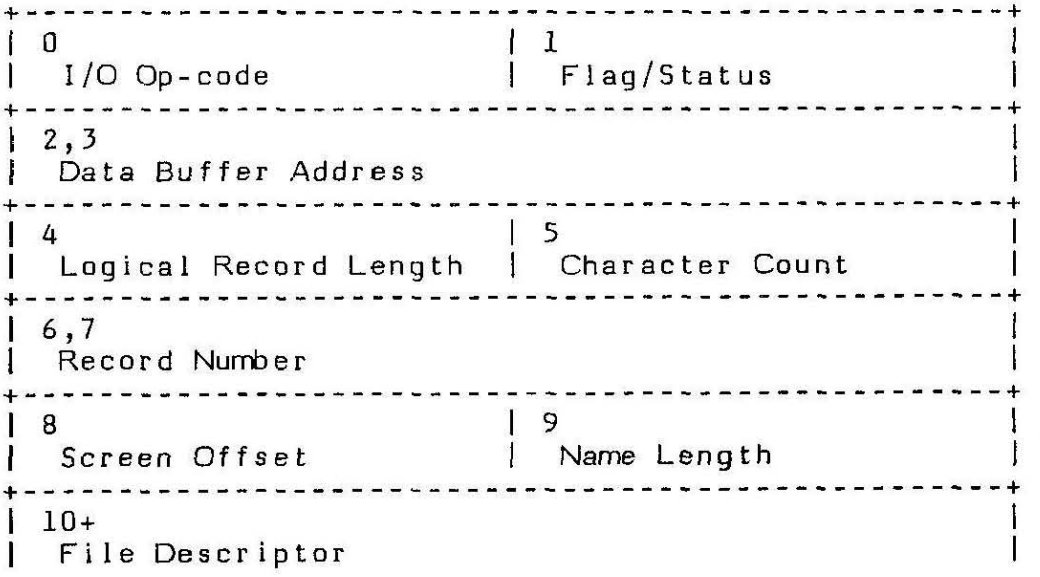
The PABs are in VDP RAM. They are created before an OPEN statement and are not released until the I/O for their corresponding peripheral has been closed.

The following describes the bytes which make up a PAB.

Byte	Bit	Contents	Meaning
0	All	I/O Op-code	The op-code for the current I/O call. See Section 18.2.1 for a description of the op-codes.
1	All	Flag/Status	All information the system needs about the file type, mode of operation, and data type. The meaning of the bits is described below.
	0-2	Error code	No error is 0. Other errors are indicated in combination with the I/O op-code. The error codes are discussed in Section 18.2.2.
	3	Record type	"Fixed length records" are 0 and "variable length records" are 1.
	4	Datatype	DISPLAY is 0 and INTERNAL is 1.
	5,6	Mode of operation	UPDATE is 00, OUTPUT is 01, INPUT is 10, and APPEND is 11.
	7	File type	"Sequential file" is 0 and "relative file" is 1.
2,3	All	Data Buffer Address	The address of the data buffer that the data must be written to or read from in VDP memory.
4	All	Logical Record Length	The logical record length for fixed length records or the maximum length for a variable length record.
5	All	Character Count	The number of characters to be transferred for a WRITE op-code or the number of bytes actually read for a READ op-code.

Byte	Bit	Contents	Meaning
6,7	All	Record Number	(Only required for a relative record type file.) The record number on which the current I/O operation is performed. The most-significant bit is ignored, so this number can be from 0 through 32767.
8	All	Screen Offset	The offset of the screen characters with respect to their normal ASCII value. This is used only by the cassette interface, which must put prompts on the screen.
9	All	Name Length	The length of the file descriptor, which starts in byte 10.
10+	All	File Descriptor	The device name and, if required, the filename and options. The length of this descriptor is given in byte 9.

The following figure summarizes the bytes which make up a PAB.



Errors that occur in input/output calls are returned in byte 1 (Flag/Status) of the PAB.

18.2.1 Input/Output Op-codes

The following describes the op-codes which can be used in byte 0 (I/O Op-code) of the PAB.

18.2.1.1 OPEN--0

The OPEN operation must be performed before any data-transfer operation except those performed with LOAD or SAVE. The file remains open until a CLOSE operation is performed. The mode of operation must be given in byte 1 (Flag/Status) of the PAB. Changing the mode of operation after an OPEN causes unpredictable results.

If a record length of 0 is given in byte 4 (Logical Record Length) of the PAB, the assigned record length (which depends on the peripheral) is returned in byte 4. If a non-zero record length is given, it is used after being checked for correctness with the given peripheral.

18.2.1.2 CLOSE--1

The CLOSE operation closes the file. If the file was opened in OUTPUT or APPEND mode, an End of File (EOF) record is written to the device or file before closing the file.

After the CLOSE operation, you can use the space allocated for the PAB for other purposes.

18.2.1.3 READ--2

The READ operation reads a record from the selected device and copies the bytes into the buffer specified in bytes 2 and 3 (Data Buffer Address) of the PAB. The size of the buffer is specified in byte 4 (Logical Record Length) of the PAB. The actual number of bytes stored is specified in byte 5 (Character Count) of the PAB. If the length of the input record exceeds the buffer size, the remaining characters are discarded.

18.2.1.4 WRITE--3

The WRITE operation writes a record from the buffer specified in bytes 2 and 3 (Data Buffer Address) of the PAB. The number of bytes to be written is specified in byte 5 (Character Count) of the PAB.

18.2.1.5 RESTORE/REWIND--4

The RESTORE/REWIND operation repositions the file read/write pointer to the beginning of the file or, in the case of a relative record file, to the record specified in bytes 6 and 7 (Record Number) of the PAB.

The RESTORE/REWIND operation can only be used if the file was opened in INPUT or UPDATE mode. For relative record files, you can simulate a RESTORE in any mode by specifying the record at which the file is to be positioned in bytes 6 and 7 (Record Number) of the PAB. The next operation then uses the indicated record.

18.2.1.6 LOAD--5

The LOAD operation loads a memory image of a file from an external device or file into VDP RAM. The LOAD operation is used without a previous OPEN operation. Note that the LOAD operation requires as much buffer in VDP RAM as the file occupies on the diskette or other device.

For a LOAD operation, the PAB needs the op-code in byte 0 (I/O Op-code), the starting address of the VDP RAM memory area into which the file is to be copied in bytes 2 and 3 (Data Buffer Address), the maximum number of bytes to be loaded in bytes 6 and 7 (Record Number), the name length in byte 9 (Name Length), and the file descriptor information in bytes 10+ (File Descriptor).

For related information, see the explanation of the RUN PROGRAM FILE option from the Editor/Assembler selection list in Section 2.5.

18.2.1.7 SAVE--6

The SAVE operation writes a file from VDP RAM to a peripheral. The SAVE operation is used without a previous OPEN operation. Note that the SAVE operation copies the entire memory image from the buffer in VDP RAM to the diskette or other device.

For a SAVE operation, the PAB needs the op-code in byte 0 (I/O Op-code), the starting address of the VDP RAM memory area from which the file is to be copied in bytes 2 and 3 (Data Buffer Address), the number of bytes to be saved in bytes 6 and 7 (Record Number), the name length in byte 9 (Name Length), and the file descriptor information in bytes 10+ (File Descriptor).

For related information, see the explanation of the SAVE utility in Section 24.5.

18.2.1.8 DELETE--7

The DELETE operation deletes the file from the peripheral. The operation also performs a CLOSE.

18.2.1.9 SCRATCH RECORD--8

The SCRATCH RECORD operation removes the record specified in bytes 6 and 7 (Record Number) from the specified relative record file. This operation causes an error for peripherals opened as sequential files.

18.2.1.10 STATUS--9

The status is in byte 8 (Screen Offset) of the PAB. The status byte returns the status of a peripheral and can be examined at any time. All of the bits have meaning if the file is currently open. Bits 6 and 7 only have meaning for files that are currently open. Otherwise, they are reset. The bits return the information shown on the next page.

Bit	Information
0	If set, the file does not exist. If reset, the file does exist. On some devices, such as a printer, this bit is never set since any file could exist.
1	If set, the file is protected against modification. If reset, the file is not protected.
2	Reserved for possible future use. Fixed to 0 by the current peripherals.
3	If set, the data type is INTERNAL. If reset, the data type is DISPLAY or the file is a program file.
4	If set, the file is a program file. If reset, the file is a data file.
5	If set, the record length is VARIABLE. If reset, the record length is FIXED.
6	If set, the file is at the physical end of the peripheral and no more data can be written.
7	If set, the file is at the end of its previously created contents. You can still write to the file (if it was opened in APPEND, OUTPUT, or UPDATE mode), but any attempt to read data from the file causes an error.

18.2.2 Error Codes

Errors are indicated in bits 0 through 2 of byte 1 (Flag/Status) of the PAB. An error code of 0 indicates that no error has occurred. However, an error code of 0 with the COND bit (bit 2) set in the STATUS byte at address >837C indicates a bad device name.

The table on the following page shows the possible error codes and their meanings.

Error Code	Meaning
0	Bad device name.
1	Device is write protected.
2	Bad open attribute such as incorrect file type, incorrect record length, incorrect I/O mode, or no records in a relative record file.
3	Illegal operation; i.e., an operation not supported on the peripheral or a conflict with the OPEN attributes.
4	Out of table or buffer space on the device.
5	Attempt to read past the end of file. When this error occurs, the file is closed. Also given for non-extant records in a relative record file.
6	Device error. Covers all hard device errors such as parity and bad medium errors.
7	File error such as program/data file mismatch, non-existing file opened in INPUT mode, etc.

18.2.3 Device Service Routine Operations

Device Service Routines (DSRs) react in specific ways to various operations and conditions. These reactions are described in the following sections.

18.2.3.1 Error Conditions

If a non-existent DSR is called, the File Management System returns with the COND bit (bit 2) set in the STATUS byte at address >837C.

If the DSR detects an error, it indicates the error in bits 0 through 2 of byte 1 of the PAB. Therefore, your assembly language program must clear these bits before every I/O operation and check them after every I/O operation.

18.2.3.2 Special Input/Output Modes

The DSR uses only the first part of the file descriptor in its search for the requested peripheral. The remainder of the descriptor can be used to indicate special device-related functions such as transmission rate, print width, etc. The DSR ignores descriptor portions that it does not recognize.

An example of a special I/O mode descriptor that sets values for the RS232 Interface is

```
RS232.BAUDRATE=1200.DATABITS=7.CHECKPARITY.PARITY=ODD
```

18.2.3.3 Default Handling

The DSR has certain defaults that are used if no values are specified. The following shows these defaults.

Possibilities	Default
Sequential or relative	Sequential.
UPDATE, OUTPUT, INPUT, or APPEND	UPDATE.
DISPLAY or INTERAL	DISPLAY.
Fixed or variable length	Fixed if relative and variable if sequential.
Logical record length	Depends on the specific peripheral.

18.2.4 Memory Requirements

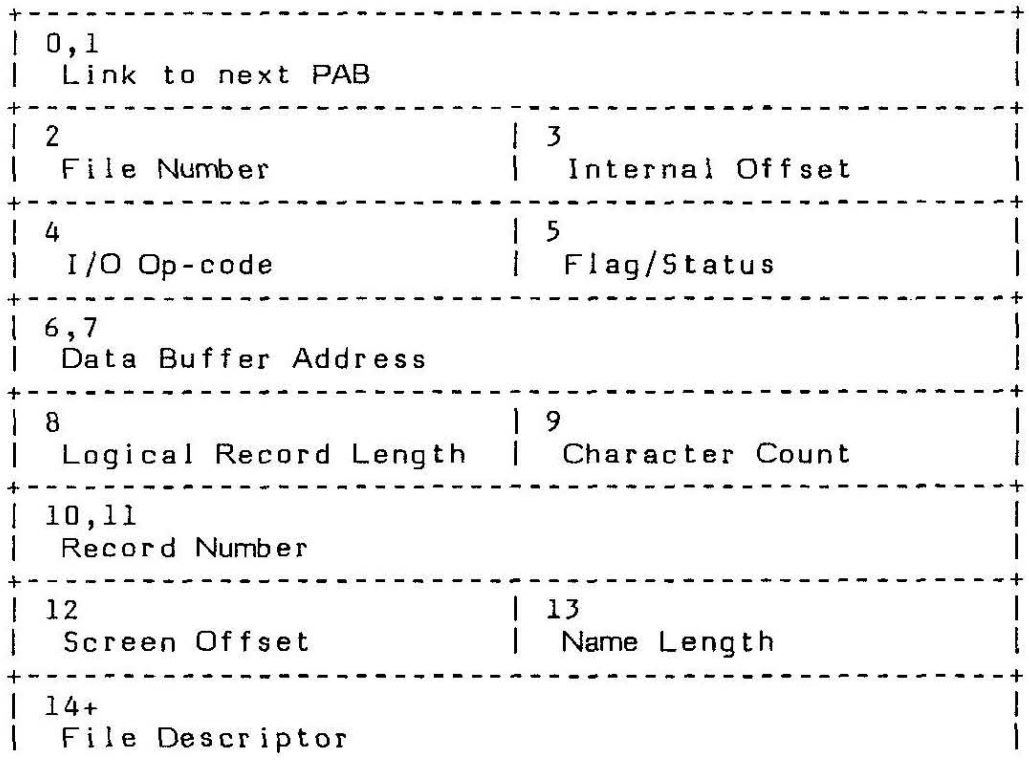
The DSR uses Registers 0 through 10 of the calling Workspace and addresses >834A through >836D. If the DSR is called in a non-interrupt driven mode (for example, through a standard DSR entry), addresses >83DA through >83DF are used. Also used are PAD (See Section 24.3.1) and VDP RAM.

18.2.5 Linkage to TI BASIC

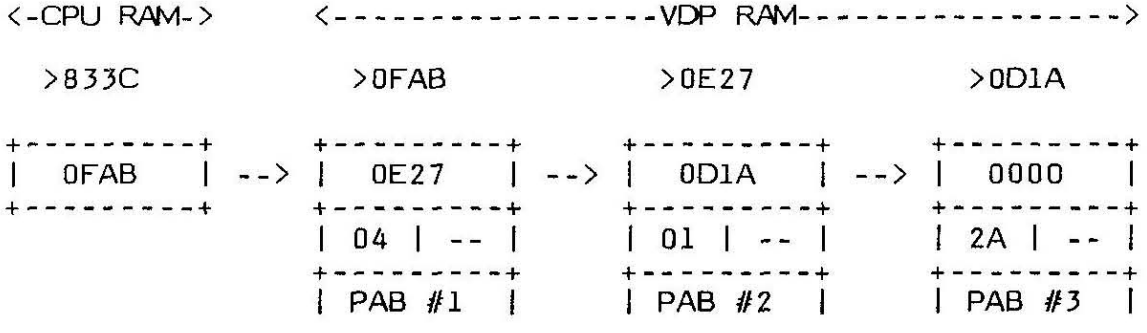
When using TI BASIC, the PAB is modified by the addition of four bytes at the beginning of the PAB. The list on the next page describes the bytes which make up a PAB when it is called from TI BASIC.

Byte	Contents	Meaning
0,1	Link to next PAB	The address of the next PAB in the chain of PABs used by TI BASIC. The last PAB in the chain has a value of >0000 in these bytes.
2	File Number	The number assigned to the file by TI BASIC.
3	Internal Offset	If 0, there is no effect. If non-zero, it is the value to be added to the start address of the data buffer before the next PRINT or INPUT operation. This is only used if the previous PRINT operation ended in a semicolon (;) or comma (,) or if the previous INPUT operation ended in a comma (,).
4	I/O Op-code	Same as byte 0 in the PAB described in Section 18.2.
5	Flag/Status	Same as byte 1 in the PAB described in Section 18.2.
6,7	Data Buffer Address	Same as bytes 2,3 in the PAB described in Section 18.2.
8	Logical Record Length	Same as byte 4 in the PAB described in Section 18.2.
9	Character Count	Same as byte 5 in the PAB described in Section 18.2.
10,11	Record Number	Same as bytes 6,7 in the PAB described in Section 18.2.
12	Screen Offset	Same as byte 8 in the PAB described in Section 18.2.
13	Name Length	Same as byte 9 in the PAB described in Section 18.2.
14+	File Descriptor	Same as bytes 10+ in the PAB described in Section 18.2.

The following figure summarizes the bytes which make up a PAB.



The following shows how three PABs might be linked in TI BASIC.



18.3 EXAMPLE OF FILE ACCESS

The following program opens a fixed 80 file called DSK1.DATA, reads a record from it, waits for you to press a key, closes the file, and returns to the calling program.

```
            DEF     DSR
            REF     DSRLNK,VMBW,VMBR,VSBW,KSCAN
*
PABBUF      EQU     >1000
PAB         EQU     >F80
*
STATUS      EQU     >837C
PNTR        EQU     >8356
*
SAVRTN      DATA    0
PDATA       DATA    >0004,PABBUF,>5000,>0000,>0009
            TEXT    'DSK1.DATA'
            EVEN
READ        BYTE    >02
CLOSE       BYTE    >01
*
MYREG       BSS     >20
BUFFER      BSS     80
*
DSR         MOV     R11,@SAVRTN     Save return address.
            LWPI    MYREG           Load own registers.
            LI      R0,PAB
            LI      R1,PDATA
            LI      R2,>20
            BLWP    @VMBW           Move PAB data into PAB in VDP RAM.
*
            LI      R6,PAB+9        Pointer to name length.
            MOV     R6,@PNTR        Store pointer to name length in >8356.
*
            BLWP    @DSRLNK         Open file.
            DATA    8
*
            MOVB    @READ,R1
            LI      R0,PAB
            BLWP    @VSBW           Change I/O op-code to read.
*
```

```
        MOV     R6,@PNTR        Restore pointer to name.
        BLWP    @DSRLNK         Read one record.
        DATA    8
*
        LI      R0,PABBUF
        LI      R1,BUFFER
        LI      R2,80
        BLWP    @VMBR           Move to CPU buffer.
*
        LI      R0,>102         Specify beginning screen location.
        LI      R1,BUFFER
        LI      R2,80
        BLWP    @VMBW           Move line to screen.
*
LOOP
        BLWP    @KSCAN          Wait for key press.
        MOVB    @STATUS,R0
        JEQ     LOOP
*
OVER    MOVB    @CLOSE,R1
        LI      R0,PAB
        BLWP    @VSBW           Change I/O op-code to close.
*
        MOV     R6,@PNTR        Restore pointer to name.
        BLWP    @DSRLNK         Close file.
        DATA    8
*
        CLR     R0
        MOVB    R0,@STATUS      So that no error is reported.
        MOV     @SAVRTN,R11     Saved return address.
        RT                      Return to calling routine.
*
        END
```

SECTION 19: THE LINKING LOADER

The Linking Loader loads assembly language programs into the Memory Expansion unit. The Loader is written in assembly language and is included in the utility programs in the Editor/Assembler Command Module. Both compressed and uncompressed tagged object code can be handled by the Loader.

The Loader is loaded into the Memory Expansion unit as part of the utility programs when you select the LOAD AND RUN option from the Editor/Assembler selection list or when a TI BASIC program executes the CALL INIT statement. It is also loaded the first time CALL LOAD is executed without the Loader in memory. The utility programs are loaded in the lower block of memory, starting at address >2000, in the Memory Expansion unit.

19.1 MEMORY ALLOCATION

The Loader always attempts to load relocatable programs into the 24K block of high memory located in the Memory Expansion unit at addresses >A000 through >FFD7. If insufficient space is there, the Loader places the program between the utilities and the REF/DEF table in the low memory of the Memory Expansion unit at approximately addresses >2676 through >3F37.

When a 0-tag (see Section 15.2 for a description of tags) is encountered, and high memory has enough space, the starting load address is updated from the First Free Address in High memory (FSTHI, equal to UTLTAB+2 or address >2024), and the module length is added to FSTHI.

If the high memory is full or has insufficient space to allocate the program, the Loader checks the low memory. If sufficient space is there, the starting load address is updated from the First Free Address in Low memory (FSTLOW, equal to UTLTAB+6 or address >2028) and the module length is added to FSTLOW.

Loading absolute code does not affect these memory pointers. Thus, loading relocatable code after absolute code has been loaded may cause the absolute code to be overwritten. Similarly, loading absolute code after relocatable code may overwrite the relocatable code.

The Loader can be called repeatedly to load more than one file until both high and low memory are full. The programs that have been loaded are accessible until one of the following conditions occurs.

- The EDIT, LOAD, or SAVE option is selected from the Editor/Assembler selection list, causing the Editor to be loaded into memory.
- The ASSEMBLE option is selected from the Editor/Assembler selection list and the Assembler program is loaded into memory.
- The LOAD AND RUN option is selected from the Editor/Assembler selection list and the file name is entered.
- The RUN PROGRAM FILE option is selected from the Editor/Assembler selection list, and a program file is loaded into memory.
- CALL INIT is executed from TI BASIC.
- An error occurs when loading a program. Then all previous loads are invalid.

19.2 THE REF/DEF TABLE

The object tags generated by DEF statements (5- and 6-tags--see Section 15.2.1 for more information on tags) define locations in a program that can be referenced by other routines. Additionally, they can define assembly language programs which can be called by name from the LOAD AND RUN or RUN options on the Editor/Assembler selection list or from the CALL LINK statement from TI BASIC.

If the entry point is specified in the label field of an END statement (tag 1 or 2), the Loader branches to the entry address and starts execution without returning to the Editor/Assembler. In this case, the screen is not cleared or changed in color, so clearing the screen and specifying a screen color is your responsibility. Also, the Workspace Registers at USRWS are not loaded, so you must load USRWS or your own Workspace Registers with the LWPI instruction.

The 5- and 6-tags contain the name and address that are associated with a DEF instruction. These names and addresses are placed in the REF/DEF table starting at the highest address in low memory (>3FFF) and going toward >3000. Each entry is eight bytes, six for the ASCII name and two for the address. Some REFs and DEFs are predefined and are placed in the REF/DEF table when the Loader is placed in memory. The REF/DEF table entries for your programs normally start at >3F38 and go toward >3000 from there. Each time an item is added to the REF/DEF table, the pointer to the Last Free Address in Low Memory (LSTLOW, equal to UTLTAB+8 or address >202A) is adjusted.

The Loader also resolves REFs in your programs. Any DEFed symbol can be REFed in your program. REFed symbols are stored in the REF/DEF table in the same way as DEFed symbols. To distinguish a REF entry from a DEF entry, the first word of REF entries is in two's-complement form.

REFs are resolved and deleted from the table as soon as the corresponding DEF is found. DEFs remain until the next time the utilities are loaded, which resets the memory pointers.

The following list gives the predefined symbols for which the Loader resolves references. The address is the address to which the symbols are equated. An error occurs when the program is executed if a reference is never defined with the DEF instruction. The use of these symbols is described in Section 16.

Name	Address	Description
UTLTAB	>2022	Start of the utility variable table.
PAD	>8300	Start of CPU scratch pad RAM.
GPLWS	>83E0	GPL interpreter workspace pointer.
SOUND	>8400	Sound chip register.
VDPRD	>8800	VDP RAM Read Data Register.
VDPSTA	>8802	VDP RAM Read Status Register.
VDPWD	>8C00	VDP RAM Write Address Register.
SPCHRD	>9000	Speech Read Data Register.
SPCHWT	>9400	Speech Write Data Register.
GRMRD	>9800	GROM/GRAM Read Data Register.
GRMRA	>9802	GROM/GRAM Read Address Register.
GRMWD	>9C00	GROM/GRAM Write Data Register.
GRMWA	>9C02	GROM/GRAM Write Address Register.
SCAN	>000E	Address of branch to the keyboard scan utility (KSCAN).

The following list gives the utilities, also discussed in Section 16, for which the Loader resolves references.

Name	Use
VSBW	Writes a single byte to VDP RAM.
VMBW	Writes multiple bytes to VDP RAM.
VSBR	Reads a single byte from VDP RAM.
VMBR	Reads multiple bytes from VDP RAM.
VWTR	Writes a single byte to a VDP Register.
KSCAN	Scans the keyboard.
GPLLNK	Links your program to Graphics Programming Language routines.
XMLLNK	Links your program to the assembly language routines in the console ROM or RAM.
DSRLNK	Links your program to Device Service Routines.
LOADER	Links your program to the Loader to load TMS9900 tagged object code.

19.3 OBJECT TAGS

Object format tags are created in object code when your assembly language code is assembled. The tags provide the Loader with the information it needs to load the object code. All tags 0 through 9 and A through I, with the exception of D, E, G, and H, can be used by the Loader. Any other tags cause the Loader to stop with an error. In addition, the colon (:) as the first character in a record signifies the end of the file. The actions taken for each of the object format tags are described below. For a further discussion of object format tags, see Section 15.2.

Code	Name	Actions
0	Module ID	The First Free Address pointer is placed in the relocation base register and the load address register. The module length is added to the First Free Address pointer. The module name is ignored.
1,2	Entry Address	A value of 1 indicates an absolute entry address. A value of 2 indicates a relocatable entry address. One of these tags may appear at the end of the object code file by specifying an entry point with an END in your assembly language program. The Loader immediately executes any object code starting with one of these tags.
3,4	External References	A value of 3 indicates that the symbol is in relocatable code. A value of 4 indicates that the symbol is in absolute code. The REFed symbol is placed in the REF/DEF table with the address of the symbol (plus the relocation base if the tag is 3). The entry is deleted from the table when the corresponding DEF is found. The first word of the reference is given in two's-complement notation.
5,6	External Definitions	A value of 5 indicates that the symbol is in relocatable code. A value of 6 indicates that the symbol is in absolute code. The DEFed symbol is placed in the REF/DEF table with the address of the symbol (plus the relocation base if the tag is 5).
7	Record Checksum	The checksum is tested with the computed value.

Code	Name	Actions
8	Ignored Checksum	The value field is ignored.
9	Absolute Load Address	The value is placed in the current address register.
A	Relocatable Load Address	The value plus the relocation base register is placed in the current address register.
B	Absolute Data	The data is placed at the address specified by the current address register. The current address is incremented by 2.
C	Relocatable Data	The value of the data plus the relocation base register is placed at the address specified by the current address register. The current address is incremented by 2.
D	Load Bias	Loading halts with an error.
E	Undefined	Loading halts with an error.
F	End of Record	The rest of the record is ignored and a new record is read.
G,H	Undefined	Loading halts with an error.
I	Program Segment ID	Ignored.
:	End of File	When the first character of a record, loading halts.

Note: When absolute code is loaded into memory, free space pointers are not updated or checked against the size of the program. Thus, the Loader does not issue an error message if the program overwrites a utility program (including the Loader itself) or does not fit in memory. When loading a short program by poking data into memory, the same situation occurs. To run the poked program, you must modify the REF/DEF table by adding a DEFed entry for the program. To avoid having the Loader overwrite memory, the free space address pointer should be adjusted so that the Loader does not load a program into an area where an AORGed or poked program is located. The TI BASIC subroutine PEEK can be used to read the pointers before they are modified. See Section 17.1.4 for more information.

19.3.1 Loader Error Codes

If no error occurs, the second bit in the STATUS byte at >837C is reset on return from the Loader. If an I/O or loading error ocurs, the status bit is set. The following are the error codes used.

Code	Meaning
0-7	Standard I/O errors.
8	Memory overflow.
9	Not used.
10	Illegal tag.
11	Checksum error.
12	Unresolved reference.

Note: If the Loader finds a label in the DEF statement that is already defined in the REF/DEF table, a duplicate definition error is issued and loading stops.

SECTION 20: SOUND

With the TI Home Computer, your program can generate up to three tones, with a range of 110 to 55,938 Hz, and one noise. Sound generation involves setting the frequency of the tone or the type of noise desired, setting the volume or attenuation, setting the duration of the tone or noise, and then starting the sound. Sound is generated using the TMS9919 Sound Generator Controller.

Three addresses in CPU RAM are associated with processing sound information. You place a pointer to the sound table in VDP RAM at address >83CC. You place >01 at address >83CE to start the processing of the sound generator. This address is used by the interrupt routine as a count-down timer during the execution of sound. The least-significant bit of the byte at address >83FD (which is the least-significant byte of GPL Workspace Register 14) must be set to indicate that the sound table is in VDP RAM.

The VDP interrupt must be disabled while you are setting up memory for sound operation. Normally the VDP interrupt is disabled when control is transferred to your assembly language program. If necessary, you can disable the VDP interrupt with the LIMI 0 instruction. After the memory has been set up, you must enable VDP interrupt with the LIMI 2 instruction in order to start processing sound operation.

Note: Interrupts should never be enabled while your program is accessing VDP memory because the interrupt routines can change the VDP read or write addresses. To execute sound lists automatically, you must enable interrupts momentarily at least every sixtieth of a second. If your program has a key scanning loop, such as the one shown below, that is often a good place to enable interrupts.

```
GETKEY  LIMI    2           Test for interrupt.
        LIMI    0
        BLWP    @KSCAN
        MOVB    @>837C,0
        JEQ     GETKEY
        .
        .
        .
```

20.1 SOUND TABLE

You must construct a sound table in order to create sounds. The sound table consists of sound lists, each of which provides the information necessary for the operation of the sound processor. For a tone, the information consists of the register value and data for frequency and attenuation. For a noise, the information consists of a noise source and attenuation. Tones can be specified, singly or in combination, for generator 1, 2, or 3. Noise can be specified by a noise generator.

When you are generating tones, the first byte in the sound list is the number of bytes to be loaded into the sound processor. Following that are the bytes to be loaded. A generator and frequency can be specified by two bytes. A generator and attenuation can be specified by one byte. A generator and noise can be specified by one byte. After you give all generator, tone, noise, and attenuation specifications, you give a duration time. A sound list consists of a series of these specifications. Each specification consists of a byte count, a series of generator, tone, noise, and attenuation specifications, and a duration time.

The two bytes that specify the tone information contain the following.

Byte	Bit	Contents
1	0	1.
	1 - 3	Operation.
	4 - 7	Four least significant frequency bits.
2	0 - 1	00.
	2 - 7	Six most significant frequency bits.

The byte that specifies attenuation information contains the following.

Byte	Bit	Contents
1	0	1.
1	1 - 3	Operation.
1	4 - 7	Attenuation.

The byte that specifies noise information contains the following.

Byte	Bit	Contents
1	0	1.
1	1 - 3	Operation.
1	4	0.
1	5	White noise or periodic noise.
1	6 - 7	Type of noise.

20.1.1 Operation Specification

The operations are described in bits 1 through 3 of byte 1.

Operation	Significance
0 0 0	Tone 1 frequency
0 0 1	Tone 1 attenuation
0 1 0	Tone 2 frequency
0 1 1	Tone 2 attenuation
1 0 0	Tone 3 frequency
1 0 1	Tone 3 attenuation
1 1 0	Noise control
1 1 1	Noise attenuation

20.1.2 Frequency Specification

The frequency value required in the Sound Table is contained in bits 2 through 7 of byte 2 and bits 4 through 7 of byte 1 of the bytes specifying tone information. This allows a value of from 00 0000 0000 through 11 1111 1111 (>000 through >3FF). This value, which is kept in the period register, defines half of the period of the desired frequency, N. The value is loaded into a 10-stage tone counter which is decremented at a rate equal to n/16, where n is the standard input clock frequency, 3.579545 MHz. When the tone counter decrements to zero, a borrow signal is produced. This borrow signal toggles the frequency flip-flop and reloads the tone counter. Thus, the period of the desired frequency is twice the value of the period register.

The frequency code, f, is equal to (n/32)/N, or K/N. The rate is specified so that the value of K is equal to 111860.8 regardless of the input clock frequency. Thus, f = 111860.8/N can always be used to calculate the frequency code. The lowest possible frequency is 110 Hz, and the highest is 55,938 Hz.

For example, to output a frequency of 110 Hz, f = 111,860.8/110 or approximately 1017 (>3F9 or 11 1111 1001 binary). Placing this value into the correct positions (most significant six bits in bits 2 through 7 of byte 2 and least significant four bits in bits 4 through 7 of byte 1), adding an operation description of 000 in bits 1 through 3 of byte 1 to specify a frequency for generator 1, and putting in the required bit values gives the correct byte values. They are 1000 1001 (>89) in byte 1 and 0011 1111 (>3F) in byte 2.

20.1.3 Attenuation Specification

The attenuation can be from 0 to 28 DB. In addition, the generator can be made silent by specifying the maximum attenuation. The attenuation is specified in bits 4 through 7 of the byte specifying attenuation. In determining the attenuation, these four bits may be regarded as having a binary 0 after them. Thus, a value of 0001 may be considered as 00010, or an attenuation of 2 DB. A value of 0110 may be considered as 01100, or an attenuation of 12 DB. Setting all the bits on, or a value of 1111, turns the sound generator off.

For example, to turn off generator 1, data 1001 1111 (>9F) is placed in the byte. To put an attenuation of 0 DB in the noise generator, data 1111 0000 (>F0) is placed in the byte.

20.1.4 Noise Specification

The noise generator consists of a noise source and an attenuator. The noise source is a 15-stage shift register with an exclusive OR feedback network. The feedback network has provisions to protect the shift register from being locked in the zero state.

The feedback network has two feedback tap configurations as determined by bit 5. If bit 5 is 0, the noise is of the periodic type. If bit 5 is 1, white noise results. (See the User's Reference Guide for more information on white noise and periodic noise.) When bit 5 is changed, the shift register is cleared.

The register shifts at one of four rates as determined by bits 6 and 7. The fixed shift rates are derived from the input clock, which has a frequency, n, of 3.579545 MHz. A value of 00 specifies a shift rate of n/512 or 6991. A value of 01 specifies a shift rate of n/1024 or 3496, and a value of 10 specifies a shift rate of n/2048 or 1738. With a value of 11, you can define the shift rate by feeding the shift rate into generator 3.

For example, to produce a white noise with a shift rate of 1748, data 1111 0110 (>F6) must be placed in the byte.

20.1.5 Duration Control

After the byte giving the number of specification bytes to be loaded and the specifications are entered, a byte which controls the duration of the sound is given. This byte specifies the length in sixtieths of a second. It can range from >00 (no time, which stops the generator) through >FF (approximately 4.25 seconds).

For example, to specify a tone with a frequency of 110 Hz and 2 DB of attenuation for 0.5 seconds on generator 1, the bytes to load are >03, >89, >3F, >91, >1E. The first byte (>03) specifies that there are three data bytes to load into the sound generator hardware. The next two bytes (>893F) specify a tone of 110 Hz on generator 1. The next byte (>91) specifies an attenuation of 2 DB on generator 1. The last byte (>1E) specifies a duration of 30/60ths of a second. You can terminate sound with data >01, >9F, 0.

20.2 DIRECT ACCESS TO THE SOUND GENERATOR

Any sound list, excluding the time duration data, can be directly loaded into the sound processor at address >8400. This address is in the DEF table in the utility program, so REF SOUND must be included in your program. If you use this direct loading, you must include an appropriate time delay so that the sound is heard for the desired period. This method does not involve the interrupt processing routine in the console.

20.3 SOUND GENERATOR FREQUENCIES

The following table lists the data used to produce various notes with generator 1. The first nybble must be changed to use generator 2 or 3.

Alternatively, you can find the note half a step above a given note by the following formula.

New frequency = old frequency * $2^{(1/12)}$

For example, the frequency half a step higher than middle A is 440 * $2^{(1/12)}$ or 466.16.

In the table, the first number after the note specifies the octave of the note. The second number, which is always 1, specifies that this is for generator 1. This table, and similar ones for generators 2 and 3, are often useful in producing music. For example, your program might contain

```
A01     EQU     >893F       A on generator 1.
A02     EQU     >A93F       A on generator 2.
A03     EQU     >C93F       A on generator 3.
A#01    EQU     >803C       A# on generator 1.
```

and so on. You may then refer to notes by name rather than by the data required to produce them.

Note	Data	Desired Frequency	Frequency Code
A01	893F	110.00	1017 (>3F9)
A#01	803C	116.54	960 (>3C0)
B01	8A38	123.47	906 (>38A)
C11	8735	130.81	855 (>357)
C#11	8732	138.59	807 (>327)
D11	8A2F	146.83	762 (>2FA)
D#11	8F2C	155.56	719 (>2CF)
E11	872A	164.81	679 (>2A7)
F11	8128	174.61	641 (>281)
F#11	8D25	185.00	605 (>25D)
G11	8B23	196.00	571 (>23B)
G#11	8B21	207.65	539 (>21B)

Note	Data	Desired Frequency	Frequency Code
A11	8C1F	220.00	508 (>1FC)
A#11	801E	233.08	480 (>1E0)
B11	851C	246.94	453 (>1C5)
C21	8C1A	261.63	428 (>1AC)
C#21	8419	277.18	404 (>194)
D21	8D17	293.66	381 (>17D)
D#21	8816	311.13	360 (>168)
E21	8315	329.63	339 (>153)
F21	8014	349.23	320 (>140)
F#21	8E12	369.99	302 (>12E)
G21	8D11	392.00	285 (>11D)
G#21	8D10	415.30	269 (>10D)
A21	8E0F	440.00	254 (>0FE)
A#21	800F	466.16	240 (>0F0)
B21	820E	493.88	226 (>0E2)
C31	860D	523.25	214 (>0D6)
C#31	8A0C	554.37	202 (>0CA)
D31	8E0B	587.33	190 (>0BE)
D#31	840B	622.25	180 (>0B4)
E31	8A0A	659.26	170 (>0AA)
F31	800A	698.46	160 (>0A0)
F#31	8709	739.99	151 (>097)
G31	8F08	783.99	143 (>08F)
G#31	8708	830.61	135 (>087)
A31	8F07	880.00	127 (>07F)
A#31	8807	932.33	120 (>078)
B31	8107	987.77	113 (>071)
C41	8B06	1046.50	107 (>06B)
C#41	8506	1108.73	101 (>065)
D41	8F05	1174.66	95 (>05F)
D#41	8A05	1244.51	90 (>05A)
E41	8505	1318.51	85 (>055)
F41	8005	1396.91	80 (>050)
F#41	8C04	1479.98	76 (>04C)
G41	8704	1567.98	71 (>047)
G#41	8304	1661.22	67 (>043)
A41	8004	1760.00	64 (>040)
A#41	8C03	1864.66	60 (>03C)
B41	8903	1975.53	57 (>039)

Note	Data	Desired Frequency	Frequency Code
C51	8503	2093.00	53 (>035)
C#51	8203	2217.46	50 (>032)
D51	8003	2349.32	48 (>030)
D#51	8D02	2489.02	45 (>02D)
E51	8A02	2637.02	42 (>02A)
F51	8802	2793.83	40 (>028)
F#51	8602	2959.96	38 (>026)
G51	8402	3135.96	36 (>024)
G#51	8202	3322.44	34 (>022)
A51	8002	3520.00	32 (>020)
A#51	8E01	3729.31	30 (>01E)
B51	8C01	3951.07	28 (>01C)
C61	8B01	4186.01	27 (>01B)
C#61	8901	4434.92	25 (>019)
D61	8801	4698.64	24 (>018)
D#61	8601	4978.03	22 (>016)
E61	8501	5274.04	21 (>015)
F61	8401	5587.65	20 (>014)

20.4 EXAMPLES

The following sections show a program segment which accesses the sound controller and two programs, one that plays a chime, and one that makes a crashing sound.

20.4.1 Accessing the Sound Controller

This program segment shows how to access the sound controller, assuming that the sound table is in VDP RAM.

```
H01     BYTE    >01
        EVEN
*
START   LIMI    0               Disable VDP interrupt.
        LI      R10,TABLE       Load sound table address in VDP
*                               RAM.
        MOV     R10,@>83CC      Load it at >83CC.
        MOVB    @H01,@>83CE     Trigger sound processing.
        SOCB    @H01,@>83FD     Set VDP RAM flag.
        LIMI    2               Enable VDP interrupt.
        .
        .
        .
```

20.4.2 A Chime

The following program repeatedly plays a chime.

```
*
* Example program to play a chime.
*
        REF     VMBW
        DEF     CHIME
*
BUFFER  EQU     >1000           VDP RAM buffer used by the sound
*                               generator.
*
H01     BYTE    >01
        EVEN
```

```
*
CHIME
        LI      R0,BUFFER       Load VDP RAM buffer address.
        LI      R1,CDATA        Pointer to the sound data.
        LI      R2,118          118 bytes to move to the VDP RAM
*                               buffer.
        BLWP    @VMBW           Move to the VDP RAM.
*
LOOP
        LIMI    0               Disable VDP interrupt.
        LI      R10,BUFFER      Load sound table address.
        MOV     R10,@>83CC      Load pointer to the table.
        SOCB    @H01,@>83FD     Set VDP flag.
        MOVB    @H01,@>83CE     Trigger sound processing.
        LIMI    2               Enable VDP interrupt.

*
LOOP2
        MOVB    @>83CE,@>83CE   Check if time is up.
        JEQ     LOOP            Repeat the sound.
        JMP     LOOP2           Wait until finished.
*
CDATA   BYTE    >05,>9F,>BF,>DF,>FF,>E3,1
        BYTE    >09,>8E,>01,>A4,>02,>C5,>01,>90,>B6,>D3,6
        BYTE    >03,>91,>B7,>D4,5
        BYTE    >03,>92,>B8,>D5,4
        BYTE    >05,>A7,>04,>93,>B0,>D6,5
        BYTE    >03,>94,>B1,>D7,6
        BYTE    >03,>95,>B2,>D8,7
        BYTE    >05,>CA,>02,>96,>B3,>D0,6
        BYTE    >03,>97,>B4,>D1,5
        BYTE    >03,>98,>B5,>D2,4
        BYTE    >05,>85,>03,>90,>B6,>D3,5
        BYTE    >03,>91,>B7,>D4,6
        BYTE    >03,>92,>B8,>D5,7
        BYTE    >05,>A4,>02,>93,>B0,>D6,6
        BYTE    >03,>94,>B1,>D7,5
        BYTE    >03,>95,>B2,>D8,4
        BYTE    >05,>C5,>01,>96,>B3,>D0,5
```

```
        BYTE    >03,>97,>B4,>D1,6
        BYTE    >03,>98,>B5,>D2,7
        BYTE    >03,>9F,>BF,>DF,0
        END
```

20.4.3 A Crash

The following program makes a crashing sound.

```
*
*  Example Program to make a crash sound.
*
        REF     VMBW
        DEF     CRASH
*
BUFFER  EQU     >1000           VDP RAM buffer used by sound
*                               generator.
*
H01     BYTE    >01
        EVEN
*
CRASH
        LI      R0,BUFFER       Load VDP RAM buffer address.
        LI      R1,CDATA        Pointer to the sound data.
        LI      R2,32           32 bytes to move to the VDP RAM
*                               buffer.
        BLWP    @VMBW           Move to VDP RAM buffer.
*
LOOP
        LIMI    0               Disable VDP interrupt.
        LI      R10,BUFFER      Load sound table address.
        MOV     R10,@>83CC      Load pointer to the table.
        SOCB    @H01,@>83FD     Set VDP RAM flag.
        MOVB    @H01,@>83CE     Start sound processing.
        LIMI    2               Enable VDP interrupt.
*
LOOP2
        MOVB    @>83CE,@>83CE   Check if time is up.
        JEQ     LOOP            Finished with the sound table.
        JMP     LOOP2           Wait until finished.
```

```
*
CDATA   BYTE    >03,>9F,>E4,>F2,5
        BYTE    >02,>E4,>F0,12
        BYTE    >02,>E4,>F2,10
        BYTE    >02,>E4,>F4,8
        BYTE    >02,>E4,>F6,6
        BYTE    >02,>E4,>F8,4
        BYTE    >02,>E4,>FA,2
        BYTE    >01,>FF,0
        END
```

SECTION 21: COLOR, GRAPHICS, AND SPRITES

The TI Home Computer gives you the capability of displaying a wide variety of colored graphics and sprites, enabling you to make your programs lively and interesting. You can place the screen in one of four modes: text, graphics, multicolor, and bit-map (available only on the TI-99/4A).

In graphics mode, you can use the standard ASCII characters and define new characters. You can make characters and their backgrounds a variety of colors. The screen is 32 columns by 24 lines. This is the mode used by the Editor/Assembler except when editing, TI BASIC, and most applications.

In multicolor mode, you can set the colors of a number of small boxes. The screen is 64 columns by 48 lines.

In text mode, you can use the standard ASCII characters and define new characters. All characters are one color, and the background is one color. The screen is 40 columns by 24 lines. This is the mode used by the Editor.

In bit-map mode (available only on the TI-99/4A because of its use of the TMS9918A video processor instead of the TMS9918 video processor), you can set any pixel (the smallest dot on the screen) on or off and make the pixels and the background a variety of colors. The screen is 256 columns by 192 lines.

In all modes except text, up to 32 sprites (moving graphics) can be created and set in motion without further program control.

21.1 VDP WRITE-ONLY REGISTERS

Before using the different modes, certain preliminary information is necessary. The following describes the eight VDP write-only registers.

VDP Register 0 — The default for Register 0 is >00 for the Editor/Assembler, TI BASIC, and TI Extended BASIC.

- Bits 0 - 5 — Reserved. Must be 000000.
- Bit 6 — Mode bit 3, called M3. If this bit is set, the display is in bit-map mode.
- Bit 7 — External video enable/disable. A value of 1 enables video input and a value of 0 disables video input.

VDP Register 1 — The default for Register 1 is >E0 in the Editor/Assembler, TI BASIC, and TI Extended BASIC. **Note:** Before changing this Register, put a copy of the new value you wish it to have at address >83D4. When a key is pressed, a copy of the value at this address is placed in Register 1.

- Bit 0 — 4/16K selection. A value of 0 selects 4K RAM operation, and a value of 1 selects 16K RAM operation.
- Bit 1 — Blank enable/disable. A value of 0 causes the active display (the entire screen) to be blank, and a value of 1 allows display on the screen. With a value of 0, the screen only shows the border color.
- Bit 2 — Interrupt enable/disable. A value of 0 disables VDP interrupt, and a value of 1 enables VDP interrupt.
- Bit 3 — Mode bit 1, called M1. If this bit is set, the display is in text mode.
- Bit 4 — Mode bit 2, called M2. If this bit is set, the display is in multicolor mode.
- Bit 5 — Reserved. Must be 0.

Bit 6 Sprite size selection. A value of 0 selects standard size sprites, and a value of 1 selects double-size sprites.

Bit 7 Sprite magnification selection. A value of 0 selects unmagnified sprites, and a value of 1 selects magnified sprites.

VDP Register 2 The default for Register 2 is >00 in the Editor/Assembler, TI BASIC, and TI Extended BASIC.

Defines the base address of the Screen Image Table. The Screen Image Table base address is equal to the value of this register times >400.

VDP Register 3 The default for Register 3 is >0E in the Editor/Assembler, >0C in TI BASIC, and >20 in TI Extended BASIC.

Defines the base address of the Color Table. The Color Table base address is equal to the value of this register times >40.

VDP Register 4 The default for Register 4 is >01 in the Editor/Assembler and >00 in TI BASIC and TI Extended BASIC.

Defines the base address of the Pattern Descriptor Table. The Pattern Descriptor Table base address is equal to the value of this register times >800.

VDP Register 5 The default for Register 5 is >06 in the Editor/Assembler, TI BASIC, and TI Extended BASIC.

Defines the base address of the Sprite Attribute List. The Sprite Attribute List base address is equal to the value of this register times >80.

VDP Register 6 — The default for Register 6 is >00 in the Editor/Assembler, TI BASIC, and TI Extended BASIC.

Defines the base address of the Sprite Descriptor Table. The Sprite Descriptor Table base address is equal to the value of this register times >800.

VDP Register 7 — The default for Register 7 is >F5 in the Editor/Assembler and >17 in TI BASIC and TI Extended BASIC.

Bits 0 - 3 — The color code of the foreground color in text mode.

Bits 4 - 7 — The color code for the background color in all modes.

The mode bits, M1, M2, and M3, are in bits 3 and 4 of Register 1 and bit 6 of Register 0. They determine the mode of the display. If they are all reset, the display is in graphics mode. If M1, in bit 3 of Register 1, is set, the display is in text mode. If M2, in bit 4 of Register 1, is set, the display is in multicolor mode. If M3, in bit 6 of Register 0, is set, the display is in bit-map mode, available only on the TI-99/4A.

21.2 GRAPHICS MODE

In graphics mode, you can use the standard ASCII characters and define patterns or characters and their foreground and background colors. The display is 32 columns by 24 lines. You can use sprites. Color and graphics are available by defining each of the 256 characters and setting their foreground and background colors. The standard ASCII characters are predefined by the system software.

21.2.1 Pattern Descriptor Table

The Pattern Descriptor Table contains descriptions of the 256 patterns or characters. By changing these descriptions, you can alter the appearance of the character on the screen. The description of each of the 256 patterns or characters takes eight bytes of information. The description of the subprogram CHAR in the User's Reference Guide discusses character definition.

In the Editor/Assembler, the Pattern Description Table starts at address >0800. Thus, the description of character >00 occupies addresses >0800 through >0807, character >01 occupies addresses >0808 through >080F, and character >FF occupies addresses >0FF8 through >0FFF.

21.2.2 Color Table

The Color Table contains the descriptions of the foreground and background colors of the characters. The most-significant four bits of the byte specify the foreground color and the least-significant four bits specify the background color. Each byte specifies the color for a group of eight characters. The 16 colors available on the TI Home Computer and their hexadecimal codes are listed on the next page.

Color	Hexadecimal Code	Color	Hexadecimal Code
Transparent	0	Medium red	8
Black	1	Light red	9
Medium green	2	Dark yellow	A
Light green	3	Light yellow	B
Dark blue	4	Dark green	C
Light blue	5	Magenta	D
Dark red	6	Gray	E
Cyan	7	White	F

In the Editor/Assembler, the Color Table starts at address >0380. Thus, the byte at address >0380 specifies the colors of characters >00 through >07, the byte at address >0381 specifies the colors of characters >08 through >0F, and the byte at address >039F specifies the colors of characters >F8 through >FF.

For example, placing a value of >17 at address >0384 sets the colors of characters >20 through >27 to black on cyan.

21.2.3 Screen Image Table

The Screen Image Table specifies the characters that occupy each of the screen positions. Each byte specifies the character at one screen position. The 768 screen positions are arranged on the screen in 24 rows of 32 columns.

In the Editor/Assembler, the Screen Image Table starts at address >0000. The first 32 addresses (>0000 through >001F) contain the characters for the first row, the second 32 addresses (>0020 through >003F) contain the characters for the second row, and so on.

For example, if the value >41 (normally the code for the ASCII character A) is at address >0022, the character described at addresses >0A08 through >0A0F of the Pattern Descriptor Table appears in the third column of the second row, assuming the Pattern Descriptor Table starts at address >0800.

21.3 MULTICOLOR MODE

In multicolor mode, the display is divided into 48 rows, each containing 64 "boxes" that are four pixels by four pixels. Each of the 3072 boxes thus defined can be one of the 16 colors available. You can use sprites in multicolor mode.

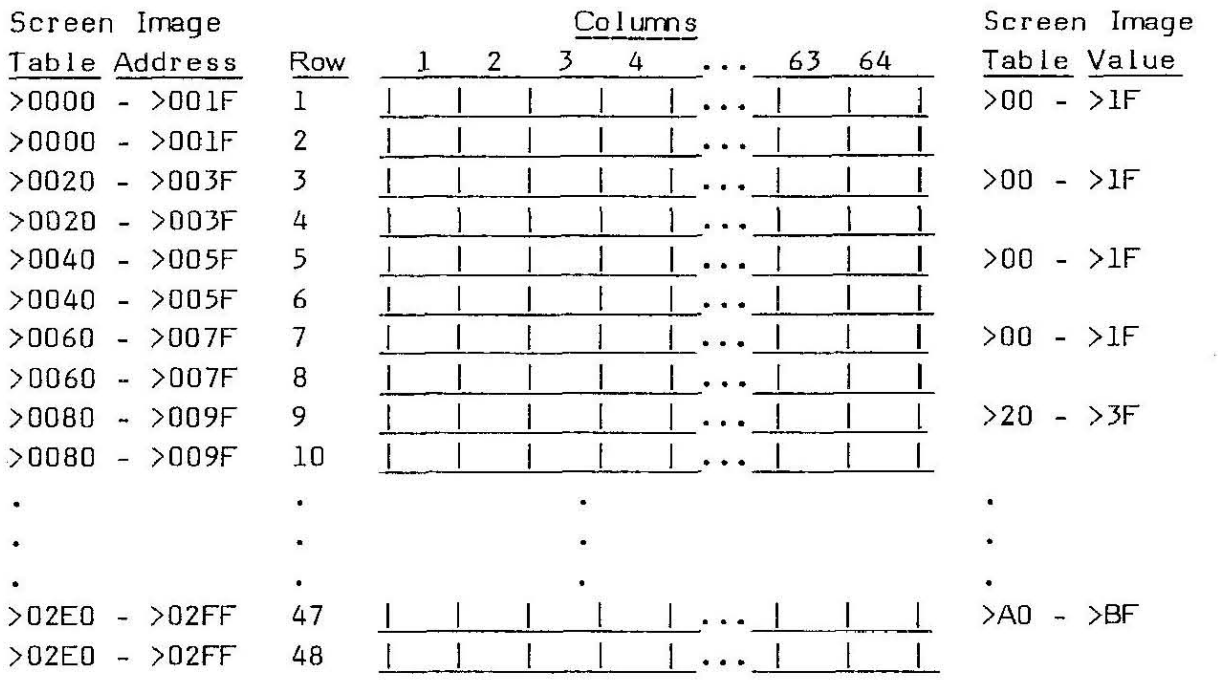
You should initialize the Screen Image Table so that the first >80 bytes contain >00 through >1F repeated four times, the next >80 bytes contain >20 though >3F repeated four times, and so on, so that the last >80 bytes contain >A0 through >BF repeated four times.

The Pattern Descriptor Table, instead of containing patterns, contains colors. Each pattern in the Pattern Descriptor Table contains eight bytes. In multicolor mode, each group of eight bytes contains 16 color descriptions, each giving the color of one box. The colors are as given in Section 21.2.2. The left four bits of each byte describe the color of one box and the right four bits describe the color of the next box on the same row.

The first byte in the Pattern Descriptor Table defines the colors of the first two boxes in the first row. The second byte defines the colors of the first two boxes in the second row. The third byte defines the colors of the first two boxes in the third row. This continues until the colors of the first two boxes in each of the first eight rows have been defined.

The next eight-byte segment similarly defines the colors of the third and fourth boxes in each of the first eight rows. This definition continues until the first 32 eight-byte segments have described all the boxes in the first eight rows. Subsequent groups of eight rows are described in a similar manner by subsequent groups of 32 eight-byte segments.

The following diagram represents the screen and how it is divided in multicolor mode. The Screen Image Table address is the offset from the beginning. The Screen Image Table value is what you should insert in the memory location.



The following table shows the Screen Image Table character code, the addresses in the Pattern Descriptor Table, assuming that it starts at address >0800, and the portions of the screen that those characters and addresses describe.

Screen Image Table Character Code	Pattern Descriptor Table Address	Row and Columns Described Row	Columns
>00	>0800	1	1 and 2
>00	>0801	2	1 and 2
>00	>0802	3	1 and 2
>00	>0803	4	1 and 2
>00	>0804	5	1 and 2
>00	>0805	6	1 and 2
>00	>0806	7	1 and 2
>00	>0807	8	1 and 2
>01	>0808	1	3 and 4
>01	>080A	2	3 and 4
.		.	.
.		.	.
.		.	.
>BF	>0DFF	48	63 and 64

21.4 TEXT MODE

In text mode, the display is 40 columns by 24 lines. You cannot use sprites. The tables used to generate the patterns are the same as the Screen Image Table and Pattern Descriptor Table used in graphics mode. However, since 960 screen positions are used instead of 768, the Screen Image Table is longer. The definitions ignore the last two bits in each entry so that each character has a 6-by-8 pixel definition. The Editor is in text mode.

The two colors available in text mode are defined in VDP write-only Register 7. The leftmost four bits describe the color of the pixels that are on and the rightmost four bits describe the color of the pixels that are off.

For example, if the Screen Image Table starts at address >0000 and >41 is at address >0202, the ASCII symbol A is placed on the 35th column of the 13th row. In graphics mode, however, this address and value would place the A on the third column of the 17th row.

21.5 BIT-MAP MODE

In the TI-99/4A Home Computer, the bit-map mode is available for defining the display. In bit-map mode, you can independently define each of the 768 (32-by-24) positions of the screen. Additionally, more color information is available for each 8-by-8 pixel pattern. You can use sprites, but not their automatic motion feature.

In the bit-map mode, the patterns that occupy screen positions are described in the Screen Image Table, the pattern descriptions are in the Pattern Descriptor Table, and the colors of the characters are described in the Color Table.

21.5.1 Screen Image Table

The Screen Image Table lists the names of the patterns, from the Pattern Descriptor Table, that are to be generated. Each name is a single byte from >00 to >FF.

The table is divided into three sections, with each section describing 256 entries. The first section of 256 entries uses descriptions taken from the first 256 entries in the Pattern Generator Table and the Color Table. The second section of 256 entries uses descriptions taken from the second 256 entries in the Pattern Generator Table and the Color Table, and the third section of 256 entries uses descriptions taken from the third 256 entries in the Pattern Generator Table and the Color Table.

The first 32 entries describe the patterns that are placed on the first row of the screen, the second 32 entries describe the patterns on the second row of the screen, and so on. The Screen Image Table should usually be placed starting at address >1800 by setting VDP write-only Register 2 to >06.

21.5.2 Pattern Descriptor Table

The Pattern Descriptor Table is divided into three sections of 256 entries each and thus contains the 768 possible patterns. Each description is eight bytes long. The description of the subprogram CHAR in the User's Reference Guide discusses character definition.

The descriptions in the first third of the table, 256 entries or 2048 bytes, describe the characters in the first third of the screen. The descriptions in the second third of the table describe the characters in the second third of the screen and the descriptions in the last third of the table describe the characters in the last third of the screen.

The Pattern Descriptor Table is >1800 bytes long. You must start it either at address >0000 or >2000 by placing either >00 or >04 in VDP write-only Register 4. If the Pattern Descriptor Table starts at address >0000, the Color Table must start at address >2000, and vice versa.

21.5.3 Color Table

The Color Table contains the descriptions of the colors of the characters in the Pattern Descriptor Table. The color codes are as described in Section 21.2.2. Eight bytes are used to describe the colors of each character. The first nybble of each byte describes the color of the pixels that are on in one row of eight pixels, and the second nybble describes the color of the pixels that are off in the same row of eight pixels.

The color descriptions in the first third of the table, 256 entries or 2048 bytes, describe the colors of the characters in the first third of the screen. The descriptions in the second third of the table describe the colors of the characters in the second third of the screen and the descriptions in the last third of the table describe the colors of the characters in the last third of the screen.

The Color Table is >1800 bytes long. You must start it either at address >0000 or >2000 by placing either >00 or >04 in VDP write-only Register 3. If the Color Table starts at address >0000, the Pattern Descriptor Table must start at address >2000, and vice versa.

21.5.4 Bit-Map Mode Discussion

In using the bit-map mode, it is usually easiest to initialize the Screen Image Table to >00 through >FF repeated three times, and then alter the entries in the Pattern Descriptor Table and the Color Table.

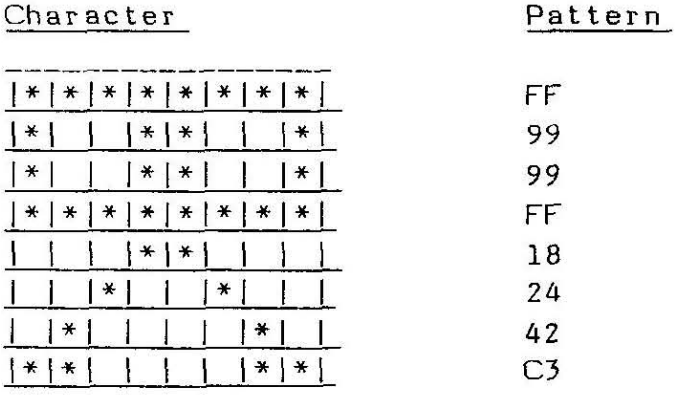
To alter a pixel on the screen, you must calculate the byte and bit to be changed in the Pattern Descriptor Table. To alter the foreground and background colors of a row of eight pixels, you must calculate the byte that must be changed in the Color Table. The following program segment allows you to find those values.

The program segment assumes that the X-value of the pixel is in Workspace Register 0 (R0) and the Y-value of the pixel is in Workspace Register 1 (R1). The offset of the byte that you must change in the Pattern Generator Table is returned in Workspace Register 4 (R4), and the bit that must be altered is returned in Workspace Register 5 (R5). The offset of the byte that you must change in the Color Table is also returned in Workspace Register 4.

```
MOV     R1,R4       R1 is the Y value.
SLA     R4,5
SOC     R1,R4
ANDI    R4,>FF07
MOV     R0,R5       R0 is the X value.
ANDI    R5,7
A       R0,R4       R4 is the byte offset.
S       R5,R4       R5 is the bit offset.
```

21.5.5 Bit-Map Mode Example

Suppose the entry for a character in the Pattern Descriptor Table is >FF9999FF182442C3. This defines the character shown below.



If the entry in the Color Table is >464646464D4D4D4D, the pattern is as follows, with B representing dark blue (>4), R representing dark red (>6), and M representing magenta (>D).

Character	Pattern	Colors
|B|B|B|B|B|B|B|B|	FF	46
|B|R|R|B|B|R|R|B|	99	46
|B|R|R|B|B|R|R|B|	99	46
|B|B|B|B|B|B|B|B|	FF	46
|M|M|M|B|B|M|M|M|	18	4D
|M|M|B|M|M|B|M|M|	24	4D
|M|B|M|M|M|M|B|M|	42	4D
|B|B|M|M|M|M|B|B|	C3	4D

On a magenta background, the magenta portions of the character blends with the background. With the pixel markings removed, the character appears as follows, with * representing dark blue (>4) and = representing dark red (>6).

```
* * * * * * * *
* = = * * = = *
* = = * * = = *
* * * * * * * *
      * *
    *     *
  *         *
* *         * *
```

21.6 SPRITES

Sprites are moving graphics that can occupy space on the screen independently and in addition to the characters which normally make up the screen. You can define and place in motion up to 32 sprites of any shape and several different sizes. After you start sprites moving, their motion continues without further program control. You can use sprites in graphics, multicolor, and bit-map mode. In bit-map mode, however, automatic motion cannot be used. Sprites are defined by setting up tables that indicate their position, their pattern, their color, their size, and their motion.

21.6.1 Sprite Attribute List

The Sprite Attribute List defines the position and color of each of the 32 possible sprites, numbered 0 through 31. As sprites move, the entries in the Sprite Attribute List are changed.

For sprites, the screen is divided into 192 (>C0) rows of 256 (>100) columns. Each of these locations is called a pixel, the smallest dot that can be displayed on the screen. The top row of pixels is designated >FF, followed by >00, >01, and so forth up to >BE. The left column of pixels is designated >00, followed by >01, >02, and so forth up to >FF.

Each sprite definition takes up four bytes in the Sprite Attribute List. The first byte is the vertical (Y) position of the sprite and starts at >FF, followed by >00 through >BE. The second byte is the horizontal (X) position of the sprite, which can be from >00 through >FF. The third byte is the pattern code, which can be from >00 through >FF. The fourth byte is the early clock attribute, which controls the location of the sprite, and color of the sprite.

Y-locations with values of >C0 through >FE are effectively off the bottom of the screen. However, a Y-location of >D0 causes that sprite and all following it in the Sprite Attribute List to be undefined. For example, if the Sprite Attribute List starts at address >0300 and no sprites are defined, the value >D0 should be placed at address >0300. If the fifth sprite is the last one active, a value of >D0 should be placed at address >0314. You can leave all 32 sprites active with the ones you do not wish to appear located off the bottom of the screen. However, it is recommended that you cause the final unused sprites to be undefined with a Y-location of >D0.

The third byte of each entry of the Sprite Attribute Table defines the character pattern to use for the sprite. The pattern can be from >00 to >FF and corresponds to a character defined in the Sprite Descriptor Table. For example, in the Editor/Assembler addresses >400 through >407 contain the entry for character >80.

The four most-significant bits in the fourth byte control the early clock of the sprite. If the last of these four bits is 0, the early clock is off. Then the sprite's location is its upper left-hand corner, and it fades in and out on the right edge of the screen. If the last of these four bits is 1, the early clock is on. Then the sprite's location is shifted 32 pixels to the left, allowing it to fade in and out on the left edge of the screen.

The color of the sprite is specified in the four least-significant bits of the fourth byte of the sprite description. The values used are the same as those given in Section 21.2.2.

In the Editor/Assembler, the Sprite Attribute List starts at address >0300. If you wish to use automatic motion, the Sprite Attribute List must start at that address. If you put the default base address (>0000) in VDP Register 6, the Sprite Descriptor Table (described in Section 21.6.2) starts at address >0000. Since the area >0000 through >03FF is used for the Screen Image Table, Color Table, and Sprite Attribute List, character codes starting at >80, at address >0400, are then normally used for sprites. When you use sprite motion, only the character codes from >80 through >EF can be used because the Sprite Motion Table starts at address >0780.

21.6.2 Sprite Descriptor Table

The Sprite Descriptor Table describes the sprites' patterns in the same way as in the Pattern Descriptor Table. However, sprites can be double-size or magnified by writing a value to the two least-significant bits in VDP Register 1. The following description tells the different sizes and magnifications possible.

Value	Description
00	Standard size sprites. Each sprite is 8 by 8 pixels, the same as a standard character on the screen.
01	Magnified sprites. Each sprite is 16 by 16 pixels, equal to four standard characters on the screen. The pattern definition is the same as for standard size sprites, but each pixel occupies four pixels on the screen.
10	Double-size sprites. Each sprite is 16 by 16 pixels, equal to four standard characters on the screen. Each sprite is defined by four consecutive patterns from the Sprite Descriptor Table. For example, each of the character codes >80, >81, >82, or >83 causes a double-size sprite to use characters >80, >81, >82, and >83 for the sprite. The first of these characters is the upper left-hand corner of the sprite, the second is the lower left-hand corner, the third is the upper right-hand corner, and the fourth is the lower right-hand corner.
11	Double-size magnified sprites. Each sprite is 32 by 32 pixels, equal to 16 standard characters on the screen. Sprites are defined as described under double-size sprites, and each pixel occupies four pixels on the screen.

In the Editor/Assembler, the Sprite Descriptor Table starts at address >0000 for pattern code >00. However, addresses >0400 and above are usually used for the block because the lower addresses are used for the Screen Image Table, Color Table, and Sprite Attribute List. The pattern defined starting at address >0400 is referred to as pattern code >80 in the Sprite Attribute Table.

21.6.3 Sprite Motion Table

The Sprite Motion Table defines the motion of sprites. It must start at address >0780. In order to move sprites, you must set up a number of conditions.

First, interrupts must be enabled during the execution of the program. Therefore, every time the program accesses the VDP RAM, interrupt handling must be disabled, which is the default. If you have enabled interrupt handling with the LIMI 2 instruction, you must disable it with a LIMI 0 instruction so that the interrupt handling routine does not alter the VDP write address.

Second, an indication of the number of sprites which have motion must be put in CPU RAM address >837A. For example, if sprites 2 and 4 are moving, the number 5 must be put at that address to allow for the motion of sprites 0, 1, 2, 3, and 4.

Third, descriptions of the motion of the sprite must be put in the Sprite Motion Table which always starts at VDP address >0780. Each sprite's motion takes up four bytes in the table. The first byte defines the vertical (Y) motion of the sprite. The second byte defines the horizontal (X) motion of the sprite. The third and fourth bytes are used by the interrupt routine.

The velocity in the first and second bytes can range from >00 to >FF. Velocities from >00 to >7F are positive velocities (down for vertical motion and right for horizontal motion), and velocities from >FF to >80 are taken as two's-complement negative velocities (up for vertical motion and left for horizontal motion).

A value of >01 causes the sprite to move one pixel every 16 VDP interrupts, or about once every 16/60ths of a second.

Since sprites are set up by loading data into VDP RAM and the TI BASIC interpreter allows interrupts, you can run sprites by successive use of the statement CALL POKEV (see Section 17.1.6). However, caution must be taken not to interfere with the TI BASIC interpreter, which does not recognize the existence of sprites. It is possible that the sprites may cause the TI BASIC interpreter to stop functioning. In TI Extended BASIC, this problem does not exist.

21.7 GRAPHICS AND SPRITE EXAMPLES

The first two of the following three assembly language programs are similar in their effect. The first places several bubble shapes on the screen and moves them up the screen. It does not use sprites, so the motion is not smooth. The second program defines the shapes as sprites, so the motion is quite smooth. In addition, pressing any key toggles the sprites from standard size to magnified sprites and back. The third program is a demonstration of automatic sprite motion.

Each of these programs must be assembled with the R option, which automatically generates Workspace Registers, and run with the LOAD AND RUN option of the Editor/Assembler.

21.7.1 Graphics Example

In the following program, several characters shaped like bubbles are placed on the screen and moved up the screen. These characters are not sprites, so the motion is not smooth. Run the program with the LOAD AND RUN option of the Editor/Assembler, using the program name BUBBLE. To leave the program, the computer must be turned off because no provision has been made for returning to the Editor/Assembler.

```
        DEF   BUBBLE
        REF   VMBW,VMBR,VSBW
*
BBLE    DATA  >3C7E,>CFDF,>FFFF,>7E3C
COLOR   DATA  >F333
BBL     BYTE  >A0
SPACE   BYTE  >A8
LOC     DATA  >01DA,>020D,>0271,>02A5,>02D6,>02E1,>0000
MYREG   BSS   >20
*
* Set up colors.
*
BUBBLE
        LWPI  MYREG
        LI    R0,>394       Color Table 20 and 21.
        LI    R1,COLOR      Load colors >F3 and >33.
        LI    R2,2          Two bytes to load.
        BLWP  @VMBW         Move to VDP RAM.
```

```
*
* Set up character definition.
*
          LI      R0,>D00      Character >A0 location.
          LI      R1,BBLE      Definition of bubble character.
          LI      R2,8         8 bytes to move.
          BLWP    @VMBW
*
* Clear screen
*
          CLR     R0           Start at VDP RAM >0000.
LOOP1     MOVB    @SPACE,R1    Move space character.
          BLWP    @VSBW        Move one space at a time.
          INC     R0           Points to next location on screen.
          CI      R0,>300      Out of screen.
          JNE     LOOP1
*
* Place bubbles on the screen.
*
          MOVB    @BBL,R1      Load character code for bubble.
          LI      R2,LOC       Load pointer to address for bubble.
LOOP2     MOV     *R2+,R0      Load real address.
          MOV     R0,R0        Check if finished loading.
          JEQ     SCROLL       Finished.  Start scrolling the screen.
          BLWP    @VSBW        Write bubble on the screen.
          JMP     LOOP2
*
* Scroll Screen.
*
VDPBF1    BSS     >20
VDPBF2    BSS     >20
*
SCROLL
          CLR     R0           VDP source address.
          LI      R1,VDPBF1    CPU buffer address.
          LI      R2,>20       Number of bytes to move.
          BLWP    @VMBR        Move >20 from VDP RAM.
*
          LI      R0,>20       VDP address >20.
          LI      R1,VDPBF2    CPU buffer address.
          LI      R2,>20       Number of bytes to move.
```

```
LOOP3   BLWP   @VMBR        Copy the line.
        AI     R0, >20      Move to lower VDP memory.
        BLWP   @VMBW        Write back to the lower line.
        AI     R0,>40       Read next line.
        CI     R0,>300      Check if end of screen.
        JL     LOOP3        If not, copy more.
*
        LI     R0,>2E0      Write the last line.
        LI     R1,VDPBF1    CPU buffer where the first line is.
        BLWP   @VMBW        Move CPU to VDP.
*
        JMP    SCROLL       Keep scrolling.
*
        END
```

21.7.2 Sprite Example

In the following program, several sprites shaped like bubbles are placed on the screen and moved up the screen. Because sprites are used, the motion is quite smooth. Run the program with the LOAD AND RUN option of the Editor/Assembler, using the program name SBBLE. To leave the program, the computer must be turned off because no provision has been made for returning to the Editor/Assembler.

```
        DEF    SBBLE
        REF    VMBW,VMBR,VSBW,VSBR
        REF    VWTR,KSCAN
*
BBLE    DATA   >3C7E,>CFDF,>FFFF,>7E3C
BBL     BYTE   >80
SPACE   BYTE   >20
SLIST   DATA   >70D0,>800F,>8068,>800F,>9888,>800F
        DATA   >A828,>800F,>B0B0,>800F,>B808,>800F
        DATA   >D000
MYREG   BSS    >20
```

```
*
* Set up character definition.
*
SBBLE
        LWPI    MYREG
        LI      R0,>400         Sprite character >80.
        LI      R1,BBLE         Definition of character.
        LI      R2,8            8 bytes to move.
        BLWP    @VMBW
*
* Define sprites.
*
        LI      R0,>300         Address of Sprite Attribute List.
        LI      R1,SLIST        Pointer to the list.
        LI      R2,26           Move 26 bytes to VDP RAM at >300.
        BLWP    @VMBW           Move the list.
*
* Scroll screen.
*
KEYBRD  EQU     >8375
STATUS  EQU     >837C
*
SET     DATA    >2000
*
SCROLL
        CLR     R5              Counter for sprite size.
LOOP
        LI      R0,>300         Pointer to the first Y-location.
READ    BLWP    @VSBR           Read one byte into R1.
        SRL     R1,8            Make it a word operation.
        CI      R1,>00D0        Check to see if it is finished.
        JEQ     KEY             Check on key input.
        DEC     R1              Decrement Y-location.
        JNE     MOVE            Move up one pixel.
        LI      R1,>00C8        Adjust the pointer.
MOVE    SLA     R1,8            Change it back to byte operation.
        BLWP    @VSBW           Write back to the list.
        AI      R0,>4           Points to the next location.
        JMP     READ            Read next Y-pointer.
```

```
*
KEY
        CLR     @KEYBRD         Clear keyboard.
        BLWP    @KSCAN          Call key scan routine.
        MOV     @STATUS,R3      Move status byte.
        COC     @SET,R3         Check for status bit.
        JEQ     CHANGE          Key pressed, change size of sprites.
        JMP     LOOP            Otherwise, keep scrolling.
*
CHANGE
        MOV     R5,R5           Is R5 null?
        JNE     SMALL           Make sprites small.
LARGE
        INC     R5              Change R5.
        LI      R0,>01E1        Change R1 to >E1.
        BLWP    @VWTR           Modify VDP register.
        JMP     LOOP            Go back to loop.
SMALL
        CLR     R5              Clear R5.
        LI      R0,>01E0        Change R1 to >E0.
        BLWP    @VWTR           Modify VDP register.
        JMP     LOOP            Go back to loop.
*
        END
```

21.7.3 Automatic Sprite Motion Example

This program is an example of using automatic motion. It places four magnified sprites in the middle of the screen and moves them in different directions at different speeds. Note that the LIMI 2 instruction is given to allow interrupts to occur. Without interrupts, sprites cannot be moved. Then, the LIMI 0 instruction is given to prevent the rest of the program from inadvertantly changing VDP RAM registers which are being used by the sprites' motion.

Run the program with the LOAD AND RUN option of the Editor/Assembler, using the program name MOVE. To leave the program, the computer must be turned off because no provision has been made for returning to the Editor/Assembler.

```
         DEF   MOVE
         REF   VMBW,VWTR,VMBR,VSBW
*
NUM      EQU   >837A
SAL      EQU   >300
COLTAB   EQU   >384
PATTN    EQU   >400
SPEED    EQU   >780
*
COLOR    DATA  >FF00
BALL     DATA  >3C7E,>FFFF,>FFFF,>7E3C
SDATA    DATA  >6178,>8006
         DATA  >6178,>8003
         DATA  >6178,>8004
         DATA  >6178,>800B,>D000
SPDATA   DATA  >0404,>0000,>F808,>0000
         DATA  >0CF4,>0000,>F0F0,>0000
MYREG    BSS   >20
*
MOVE
         LWPI  MYREG       Load my own registers.
         LI    R0,COLTAB
         MOVB  @COLOR,R1
         BLWP  @VSBW       Load background color as white.
*
         LI    R0,PATTN
         LI    R1,BALL
         LI    R2,8
         BLWP  @VMBW       Load ball pattern.
*
         LI    R0,SAL
         LI    R1,SDATA
         LI    R2,17
         BLWP  @VMBW       Load Sprite Attribute List.
*
         LI    R0,SPEED
         LI    R1,SPDATA
         LI    R2,16
         BLWP  @VMBW       Load speed of sprites.
```

```
*
        LI      R0,>81E1
        BLWP    @VWTR           Load VDP Register to magnify sprites.
*
        LI      R1,4
        SLA     R1,8
        MOVB    R1,@NUM         Specify number of sprites.
*
LOOP
        LI      R0,SAL
        LI      R3,4            Repeat 4 times.
        LI      R2,2
*
LOOP2
        LIMI    2               Enable interrupt.
        LIMI    0               Disable interrupt.
*
        LI      R1,MYREG+14     Read it into Register 7.
        BLWP    @VMBR
        AI      R7,-24
        CI      R7,>B8C8        Check if 0 < Y < 184, 24 < X < 224.
        JH      ADJUST
NEXT    AI      R0,4            Look at next sprite.
        DEC     R3
        JEQ     LOOP
        JMP     LOOP2
ADJUST  LI      R1,SDATA        Reload Sprite Attribute List.
        LI      R2,2
        BLWP    @VMBW
        JMP     NEXT
        END
```

SECTION 22: SPEECH

With the TI Solid State Speech™ Synthesizer and the TI Memory Expansion unit attached to the computer, your assembly language programs can include speech. (See the owners manuals for these products for connection instructions.)

To have the computer speak a word or phrase from the resident Speech Synthesizer vocabulary, your program must check that the Speech Synthesizer is present, obtain the address of the word or phrase to be spoken, load the address in the synthesizer so that it can be spoken, and give the command to speak. You can obtain the address by looking it up in the Section 24.6, or you can find it with your program.

You can also provide speech data directly to the Speech Synthesizer instead of using a word from the resident vocabulary. Then you use a different command to have the word or phrase spoken.

22.1 PRELIMINARY INFORMATION

Before using the Speech Synthesizer, you must be familiar with the timing requirements, addresses, and commands to use, and know how to load speech addresses, read data, and check to see that the Speech Synthesizer is attached.

22.1.1 Timing Considerations

The Speech Synthesizer requires time for each of its commands to be executed. The delay time necessary from the time a read data or read status command is given until the data or status is available is 12 microseconds. The delay time from writing external data until the next access is 10 microseconds. The delay time from loading an address until the next command is 42 microseconds.

During the delay after a read, the 8-bit peripheral bus which is connected to the Memory Expansion unit cannot be used. Therefore, the reads of data from the Speech Synthesizer and the delays must be in the area accessed by the 16-bit bus. One area convenient for this starts at address >8328.

The following program segment shows how to place the necessary delay in the proper place.

```
          REF     SPCHRD
*
SPDATA    EQU     >8328
READIT    EQU     >8330
*
CODE      MOVB    @SPCHRD,@SPDATA
          NOP
          NOP
          NOP
          RT
CLEN      EQU     $-CODE
START
* Put read routine on 16-bit bus.
          LI      R1,READIT
          LI      R2,CODE
          LI      R3,CLEN
ST2       MOV     *R2+,*R1+
          DECT    R3
          JH      ST2
```

Then to read a byte of speech data, use the instruction BL @READIT. The data is returned at the address SPDATA.

A delay of 12 or 42 microseconds can be accomplished by branching to routines, using NOP instructions, and returning. The following code segment shows the routines to delay for 12 and 42 microseconds. They can be called with BL @DLY12 and BL @DLY42 instructions respectively.

```
DLY12     NOP
          NOP
          RT
*
DLY42     LI      R1,10
DLY42A    DEC     R1
          JNE     DLY42A
          RT
```

In the rest of this section, the necessary delays are indicated by a comment as follows. Section 22.2.2 gives a complete program using these methods of delay.

```
*
* Delay as described in Section 22.1.1.
*
```

22.1.2 Addresses

The two addresses used to read and write speech data are SPCHRD at address >9000 and SPCHWT at >9400. You make them available by putting them in a REF statement at the beginning of your program.

22.1.3 Commands

You can give the Speech Synthesizer commands by placing a value in the SPCHWT address. If you load >10, you can read a byte from the SPCHRD address. If you load >4X, where X is a nybble to be loaded, the nybble is loaded. This is used to load speech addresses as described in Section 22.1.4. If you load >50, the word or phrase whose address you have loaded is spoken. If you load >60, you can load the word or phrase data, which you have constructed, to be spoken. Other commands are described in the TMS5200 Voice Synthesis Processor Data Manual, available from Texas Instruments.

22.1.4 Loading Speech Addresses

You load the address of the resident speech data by loading five nybbles of data. The first nybble is always >0, with the next four nybbles equal to the address. You load each nybble with the >4X command, where the X is replaced by the nybble you wish to load. The least significant nybble is loaded first. The fifth nybble to be loaded is always >0 so the fifth byte loaded is >40, which marks the end of the address. The following program segment demonstrates how to load the address given in location PHROM.

```
            REF     SPCHWT
PHROM
            DATA    <data>
*
LOAD
            MOV     @PHROM,R0       Address to load.
            LI      R2,4            Four nybbles to load.
LOADLP
            SRC     R0,4            Start with least significant nybble.
            MOV     R0,R1
            SRC     R1,4
            ANDI    R1,>0F00        Get only particular nybble.
            ORI     R1,>4000        Put in >4X00 format.
            MOVB    R1,@SPCHWT      Write the nybble.
            DEC     R2
            JNE     LOADLP
            LI      R1,>4000
            MOVB    R1,@SPCHWT      Write the fifth nybble.
```

In the rest of this section, the above process is indicated by a comment as follows.

```
*
* Load address as described in Section 22.1.4.
*
```

22.1.5 Reading Data

Data can be read from the Speech Synthesizer by loading the correct address using the 4X command, writing a read command (>10) at SPCHWT, and reading data from SPCHRD. The following program segment shows how to read one word of data and store it at DATAAD.

```
          REF     SPCHWT,SPCHRD
*
PHROM     DATA    <Speech Synthesizer ROM address>
DATAAD    DATA    >0000                 Data storage may be initialized to
*                                       zero.
SPDATA    EQU     >8328
READIT    EQU     >8330
H10       BYTE    >10
          EVEN
*
READ
*
* Load address as described in Section 22.1.4.
*
          MOVB    @H10,@SPCHWT          Read command.
*
* Delay as described in Section 22.1.1.
*
          BL      @READIT               Read one byte from the Speech
*                                       Synthesizer.
          MOVB    @SPDATA,@DATAAD
          MOVB    @H10,@SPCHWT          Read command again.
*
* Delay as described in Section 22.1.1.
*
          BL      @READIT               Store it in the next byte.
          MOVB    @SPDATA,@DATAAD
```

22.1.6 Checking to See if the Speech Synthesizer is Attached

Before your program attempts to produce speech, it must determine whether the Speech Synthesizer is attached. The Speech Synthesizer addresses start at >0000. If this location contains >AA, the Speech Synthesizer is attached. The following program segment checks to see if the speech unit is attached. The program segment assumes that label RUN is where to go if the Speech Synthesizer is connected and label NOT is where to go if it is not connected.

```
          REF     SPCHWT,SPCHRD
*
PHROM     DATA    >0000           Pointer to speech data.
H10       BYTE    >10             Read data command.
HAA       BYTE    >AA
          EVEN
READIT    EQU     >8330
*
          CLR     @PHROM          Look at >0000 first.
*
* Load address as described in Section 22.1.4.
*
          MOVB    @H10,@SPCHWT    Read data command.
*
* Delay as described in Section 22.1.1.
*
          BL      @READIT         Read one byte.
          CB      @SPDATA,@HAA    Is it >AA?
          JEQ     RUN             Run the program.
          JMP     NOT
```

22.2 SPEECH EXAMPLES

To speak a word or phrase from the resident vocabulary, specify its address. You can find the address by looking it up in the Appendix or with your program. After you have obtained the address, load it using the >4X command as described in Section 21.1.4, and then use the >50 command to have it spoken.

The addresses of all of the words and phrases are listed in the Appendix. For example, the address of the word HELLO is >351A. The following program segment causes the computer to say HELLO if the Speech Synthesizer is attached.

22.2.1 Accessing Speech using the Address from the Appendix

```
            REF     SPCHWT,SPCHRD
*
HELLO       EQU     >351A               Speech data address from the
*                                       Appendix.
READIT      EQU     >8330
PHROM       DATA    >0000
H50         BYTE    >50
            EVEN
*
WAIT        BL      @READIT             Read from speech read data.
            MOVB    @SPDATA,R0
            COC     @H8000,R0           Check on Status.
            JEQ     WAIT                Wait until finished speaking.
*
            LI      R0,HELLO            Load address of HELLO.
            MOV     R0,@PHROM
            LI      R2,4                Four nybbles to load.
*
* Load address as described in Section 22.1.4.
*
*
* Delay as described in Section 22.1.1.
*
            MOVB    @H50,@SPCHWT        Write execute speak command.
*
* Delay as described in Section 22.1.1.
*
```

22.2.2 Accessing Speech Directly and by Finding the Speech Address

You can also find speech data addresses in the Speech Synthesizer by searching a table, located at the beginning of the Speech Synthesizer's memory, for the word or phrase you need. The following list describes the information for each word or phrase included in the table.

Item	Length	Example
ASCII Length	1 byte	>4
Word or Phrase in ASCII	n bytes	>77 >79 >82 >69 ("MORE")
Less-Than Pointer	2 bytes	>04B6
Greater-Than Pointer	2 bytes	>0EB5
Speech Data Pointer	3 bytes	>004642
Speech Data Length	1 byte	>51

The less-than pointer gives the table address of the information related to the word or phrase next in alphabetical order. The greater-than pointer gives the table address of the information related to the preceding word or phrase in alphabetical order.

In addition, you can load speech data directly in a program and have it spoken using the >60 command.

In the following program, the entry point START looks up the address of the word "HELLO" in the Speech Synthesizer. The word is then spoken until a key is pressed. Control is then returned to the calling program.

The entry point DIRECT is an example of providing the Speech Synthesizer with direct data. When DIRECT is used, a phrase is spoken once. After it is spoken, control is returned to the calling program.

```
         REF   SPCHWT,SPCHRD,GPLWS,KSCAN
         DEF   START,DIRECT
PHROM    DATA  0
DATAAD   DATA  0
SPDATA   EQU   >8328
READIT   EQU   >8330
RSA      DATA  0
SSA      DATA  0
H8000    DATA  >8000
H4000    DATA  >4000
HELLO    EQU   >351A
SPEECH   DATA  118
         BYTE  166,209
         BYTE  198,37,104,82,151
         BYTE  206,91,138,224,232
         BYTE  116,186,18,85,130
         BYTE  204,247,169,124,180
         BYTE  116,239,185,183,184
         BYTE  197,45,20,32,131
         BYTE  7,7,90,29,179
         BYTE  6,90,206,91,77
         BYTE  136,166,108,126,167
         BYTE  181,81,155,177,233
         BYTE  230,0,4,170,236
         BYTE  1,11,0,170,100
         BYTE  53,247,66,175,185
         BYTE  104,185,26,150,25
         BYTE  208,101,228,106,86
         BYTE  121,192,234,147,57
         BYTE  95,83,228,141,111
         BYTE  118,139,83,151,106
         BYTE  102,156,181,251,216
         BYTE  167,58,135,185,84
         BYTE  49,209,106,4,0
         BYTE  6,200,54,194,0
         BYTE  59,176,192,3,0
         BYTE  0
```

```
STRING  BYTE    5
        TEXT    'HELLO'
H50     BYTE    >50
H10     BYTE    >10
H60     BYTE    >60
HAA     BYTE    >AA
        EVEN
CODE    MOVB    @SPCHRD,@SPDATA
        NOP
        NOP
        NOP
        RT
CLEN    EQU     $-CODE
DIRECT
* Put read routine on 16-bit bus.
        LI      R1,READIT
        LI      R2,CODE
        LI      R3,CLEN
DR2     MOV     *R2+,*R1+
        DECT    R3
        JH      DR2
        LI      R2,SPEECH       Pointer to speech data.
        MOV     *R2+,R3         Size.
        LI      R1,16           16 bytes to load.
        MOVB    @H60,@SPCHWT    Speak external command.
        BL      @DLY12
LOOPR
        MOVB    *R2+,@SPCHWT    Send a byte.
        DEC     R3
        JEQ     OUT             Finished with data.
        DEC     R1
        JNE     LOOPR
LOOPB
        BL      @READIT
        MOVB    @SPDATA,R0
        COC     @H4000,R0
        JNE     LOOPB           Speech not terminated; wait more.
        LI      R1,8            Queue half finished; send more data.
        JMP     LOOPR
```

```
*
START
* Put read routine on 16-bit bus.
          LI      R1,READIT
          LI      R2,CODE
          LI      R3,CLEN
ST2       MOV     *R2+,*R1+
          DECT    R3
          JH      ST2
* Check for existence of Speech Synthesizer.
          BL      @THERE
          DATA    OUT
* Start looking after the validation byte.
          INC     @PHROM
SEARCH    LI      R3,STRING
          MOVB    *R3+,R7         Length of target string.
          SRL     R7,8
          JEQ     OUT             Null string.
*
          BL      @LOAD           Load speech data address.
          BL      @DLY42
          MOVB    @H10,@SPCHWT
          BL      @DLY12
          BL      @READIT         Read length byte of string.
          MOV     @SPDATA,R4
          SRL     R4,8            Make it a word.
          INC     @PHROM
NEXT
          BL      @LOAD           Address of next letter.
          BL      @DLY42
          MOVB    @H10,@SPCHWT
          BL      @DLY12
          BL      @READIT         Read the letter.
          CB      *R3+,@SPDATA    Does it match?
          JEQ     MATCH           Yes.
          JH      HIGH            Too high.
          JMP     LOW             Too low.
```

```
MATCH
        INC     @PHROM
        DEC     R4
        JNE     STRN            More letters available.
        DEC     R7
        JEQ     SPEAK           Found the word.
HIGH
        LI      R8,2            Skip one pointer.
        JMP     NXTPHR
STRN    DEC     R7
        JNE     NEXT
LOW
        CLR     R8              Do not skip pointer.
NXTPHR
        A       R4,@PHROM       Skip over rest of word.
        A       R8,@PHROM       (Maybe) skip first pointer.
        BL      @READ           Read two bytes.
        MOV     @DATAAD,R5
        JEQ     OUT             Word not found.
        MOV     R5,@PHROM
        JMP     SEARCH          Keep looking.
*
SPEAK
        LI      R8,5
        A       R8,@PHROM
        BL      @READ           Read address of speech data.
        BL      @DLY12
        MOV     @DATAAD,@PHROM
        BL      @LOAD           Load address of speech data.
        BL      @DLY42
        MOVB    @H50,@SPCHWT    Speak word.
        BL      @WAIT           Wait for finish (or exit).
        B       @START          Do it all again.
```

```
WAIT
            MOV     R11,R10             Save return address.
            BL      @READIT             Read the status byte.
            MOV     @SPDATA,R0
            BLWP    @KSCAN              Scan keyboard.
            MOVB    @>837C,R1           Was a key pressed?
            JNE     OUT                 Yes.  Exit.
            COC     @H8000,R0           Word finished?
            JEQ     WAIT                No.  Keep waiting.
            B       *R10                Yes.  Return.
OUT         LWPI    GPLWS
            B       @>6A                Return to calling program.
* Read a word of data.
* Address in PHROM, data returned in DATAAD.
* Call with BL
READ        MOV     R11,@RSA            Save return address.
            BL      @LOAD               Load the address.
            BL      @DLY42              Wait.
            MOVB    @H10,@SPCHWT
            BL      @DLY12
            BL      @READIT             Read first byte.
            MOVB    @SPDATA,@DATAAD
            MOVB    @H10,@SPCHWT
            BL      @DLY12
            BL      @READIT             Read second byte.
            MOVB    @SPDATA,@DATAAD+1
            MOV     @RSA,R11            Return.
            RT
* Check to see if the Speech Synthesizer is attached.
* Called as BL      @THERE
                    DATA      <not there>
THERE       MOV     R11,@RSA            Save return address.
            CLR     @PHROM              Location 0.
            BL      @LOAD
            BL      @DLY42
            MOVB    @H10,@SPCHWT
            BL      @DLY12
            BL      @READIT             Read the byte.
            CB      @SPDATA,@HAA        Is is >AA?
            JEQ     RUN                 Yes.  The peripheral is attached.
```

```
* Not there.
        MOV     @RSA,R11
        MOV     *R11,R11            Fetch data word as alternate return.
        RT
* There
RUN     MOV     @RSA,R11
        INCT    R11                 Skip alternate return.
        RT
* Load address.
* Called as BL      @LOAD    with address in PHROM.
* Uses R0, R1, and R2.
LOAD    MOV     @PHROM,R0
        LI      R2,4
LOADLP  SRC     R0,4
        MOV     R0,R1
        SRC     R1,4
        ANDI    R1,>0F00            Pick off four bits.
        ORI     R1,>4000            Make it >4X.
        MOVB    R1,@SPCHWT
        DEC     R2                  Do it four times.
        JNE     LOADLP
        LI      R1,>4000            Write the >40.
        MOVB    R1,@SPCHWT
        RT
DLY12   NOP                         Short delay.
        NOP
        RT
DLY42   LI      R1,10               Long delay after address set up.
DLY42A  DEC     R1
        JNE     DLY42A
        RT
        END
```

SECTION 23: THE DEBUGGER

The Debugger program allows you to find errors in your program while it is actually running. You can read values in memory, write new values, inspect Workspace Registers and alter their values, move memory from one location to another, perform hexadecimal arithmetic, and a variety of other functions. Each function is easily available with a single-letter command.

With many of the commands, you can enter a G or V after an operand to indicate that the operation is to take place in the Command Module GROM or VDP RAM, respectively, instead of in CPU RAM or ROM.

You can define up to three bias characters, labeled X, Y, and Z. When one of these characters is appended to an operand, its value is added to the value of the operand. Thus, if you have set X to >12A and enter an operand of >1B2X, the operand used is the sum of >12A and >1B2 or >2DC.

The Debugger is located on the Editor/Assembler diskette labeled Part A. In addition to the compressed object file, called DEBUG, the entire source listing of the Debugger is included. DEBUGS is the main program, consisting mostly of COPY directives; DEBUGA is the first 400 lines of the Debugger; DEBUGB is the second 400 lines; DEBUGC is the third 400 lines; DEBUGD is the fourth 400 lines; and DEBUGE is the fifth 400 lines. You should make a backup copy of these files.

The Debugger program is a modified version of the TIBUG program designed to be used on a TI Home Computer. The Debugger is relocatable and can be loaded by the LOAD AND RUN option on the Editor/Assembler or with CALL LOAD from TI BASIC. The name of the file to load is DSK1.DEBUG if the diskette labeled Part A is in Disk Drive 1. Since the Debugger is relocatable, it is suggested that you first load the program to be debugged and then load the Debugger. Then enter the Debugger and do the required set up so that you can return to the Debugger when desired. The Loader places the Debugger program in memory the same way any other assembly language program is loaded.

Follow these steps to access the Debugger from the Editor/Assembler.

1. Insert the Editor/Assembler Command Module into the console.

2. Press any key to make the master selection list appear. Select the Editor/Assembler.

3. Insert the Editor/Assembler diskette that contains the Debugger program (Part A) into Disk Drive 1.

4. Select LOAD AND RUN from the Editor/Assembler selection list.

5. Enter the file name of the program to be debugged. If there is no such program, go to the next step.

6. Type DSK1.DEBUG (the file name of the Debugger), and press <return>.

7. Press <return> again to advance to the next prompt.

8. Enter DEBUG as the program name of the Debugger.

9. The Debugger is loaded and ready to accept commands.

Follow these steps to use the Debugger from TI BASIC.

1. Insert the Editor/Assembler Command Module into the console.

2. Press any key to make the master selection list appear. Select TI BASIC.

3. Insert the Editor/Assembler diskette that contains the Debugger program (Part A) into Disk Drive 1.

4. Load the program that you wish to debug, using the CALL LOAD statement.

5. Execute CALL LOAD ("DSK1.DEBUG") from TI BASIC.

6. Execute CALL LINK("DEBUG").

7. The Debugger is loaded and ready to accept commands.

Note: If the Debugger is entered from TI BASIC, you should immediately select the U command so that the display is correct.

23.1 PRELIMINARY INFORMATION

Pressing single-letter commands executes the Debugger routines. The Debugger automatically places a space after the letter, although you cannot see it because no cursor appears on the screen. Do not press <space> following the command letter unless you intend to terminate the command.

After choosing the command, you enter the command's operands, which consist of up to three hexadecimal fields, depending on the command. Operands contain four hexadecimal digits each. If you enter more than four digits, only the last four are used. If you enter fewer than four digits, they are considered by the Debugger to be the right-most digits of the operand.

The following are the Debugger commands.

Command	Letter	Section
Load Memory with ASCII	A	23.2
Breakpoint Set/Clear	B	23.3
CRU Inspect/Change	C	23.4
Execute	E	23.5
Find Word or Byte	F	23.6
GROM Base Change	G	23.7
Inspect Screen Location	I	23.8
Find Data Not Equal	K	23.9
Memory Inspect/Change	M	23.10
Move Block	N	23.11
Compare Memory Blocks	P	23.12
Quit Debugger	Q	23.13
Inspect or Change WP, PC, and SR	R	23.14
Execute in Step Mode	S	23.15
Trade Screen	T	23.16
Toggle Offset to and from TI BASIC	U	23.17
VDP Base Change	V	23.18
Inspect or Change Registers	W	23.19
Change Bias	X, Y, or Z	23.20
Hexadecimal to Decimal Conversion	>	23.21
Decimal to Hexadecimal Conversion	.	23.22
Hexadecimal Arithmetic	H	23.23

Each command description consists of the following information.

- A heading, consisting of the command name and command letter
- The syntax definition
- An example of the command
- The definition of the command

In the syntax definition, the following notational conventions are used. Items surrounded by <angle brackets> represent keys or items that you must provide. Items surrounded by {braces} indicate that you must choose between the two or more items included. Items surrounded by [brackets] indicate optional material. The elipsis (...) indicates that the previous item may be repeated.

Note: In the Debugger, the <escape> key is not the same as the <esc> key used in the rest of the Editor/Assembler. The <escape> key is SHIFT X on the TI-99/4 and FCTN X on the TI-99/4A.

In the example and definition of the commands, the information you type is underlined.

23.2 LOAD MEMORY WITH ASCII--A

Syntax definition:

A<start address><return>
<ASCII string><escape>

Example:

A 1000<return>
1000 NEW WORDS<escape> Places the ASCII string NEW WORDS in memory starting at address >1000.

Definition:

Places the ASCII string in memory starting at the given hexadecimal address. Any information, including <return>, that is typed before <escape> is placed in memory. After each 16 (>10) bytes, the current memory location is shown on the screen.

23.3 BREAKPOINT SET/CLEAR--B

Syntax definitions:

- To set a breakpoint:

  B<address><return>

- To clear a breakpoint:

  B<address><->

- To show breakpoints:

  B<return>

- To clear all breakpoints:

  B<->

Example:

B A000<return> Sets a breakpoint to be performed at address >A000.

Definition:

Sets a breakpoint to be performed at the given address if B is followed by an address and <return>. Clears a breakpoint if B is followed by an address and <->. Shows all breakpoints if B is followed by <return>. Clears all breakpoints if B is followed by <->.

From 11 to 16 breakpoints can be set at the same time, depending on your computer. When a breakpoint is encountered during execution, the workspace pointer, program counter, and status are saved, the breakpoint is cleared, and the Debugger is entered.

Setting a breakpoint replaces the contents of the breakpoint address with an XOP 1 call (>2F40) or a branch to BENTRY (two words). If your console does not support XOP 1, the Debugger automaticaly uses the two-word BENTRY and provides the message

BKPT USES 2 WORDS

Some TI-99/4A computers support XOP 1 and some do not. If you get the message BKPT USES 2 WORDS when you enter a breakpoint, yours does not.

If your computer uses two-word breakpoints, you must observe several precautions. First, you cannot set breakpoints at consecutive words. If you attempt to do so, the following message is displayed.

ILLEGAL CONSECUTIVE BREAKPOINT

Second, under some circumstances the computer may interpret the address of the Debugger as an instruction. For instance, suppose your program reads as follows.

```
ADDR    JMP     START   Code 1010
LOOP    MOV     R1,R2   Code C001
        DEC     R3      Code 0603
        JGT     LOOP    Code 15FD
        .
        .
        .
START   LI      R3,6    Code 0203
                        0006
        JMP     LOOP    Code 10F0
```

If you then set a two-word breakpoint at ADDR (assuming that the address of the Debugger entry is >B062), the program becomes

```
ADDR    BLWP    @BENTRY Code 0420
LOOP                    Code B062
        DEC     R3      Code 0603
        JGT     LOOP    Code 15FD
        .
        .
        .
START   LI      R3,6    Code 0203
                        0006
        JMP     LOOP    Code 10F0
```

When the program executes the JGT LOOP instruction, it interprets the address of BENTRY as an AB instruction. This situation can be avoided by not inserting breakpoints where the second word is a label.

In two-word breakpoints the previous contents of the memory are saved. Clearing a breakpoint restores the original contents of the memory location. If a breakpoint is set more than once at the same location, the following message is displayed.

ILLEGAL CONSECUTIVE BREAKPOINT

The show breakpoint command, B<return>, gives the addresses of the current breakpoints in the order in which they were set.

23.4 CRU INSPECT/CHANGE--C

Syntax definition:

C<base address>{<space> or <,>}<bit count><return>
[<data>]{<return> or <space>}

Example:

C 1380,1<return> Gives the value of the least-significant CRU bit at address >1380.

Definition:

Gives the value of the specified number of CRU bits. If you specify 1 through 15 bits, that many bits are returned as a hexadecimal number. If you specify 0 as the bit count, 16 bits are returned.

Each CRU bit takes up two bytes on the CRU. For example, supppose the CRU is as shown in the following table.

CRU Bit	Value	CRU Address
0	1	>1380
1	0	>1382
2	1	>1384
3	1	>1386
4	0	>1388
5	1	>138A
6	1	>138C
7	0	>138E
8	1	>1390
9	0	>1392
A	0	>1394
B	0	>1396
C	1	>1398
D	0	>139A
E	1	>139C
F	0	>139E

Then if you give the command

C 1380,1

The Debugger returns 0001. If you give the command

C 1380,5

The Debugger returns 000D. If you give the command

C 1380,8

The Debugger returns 006D. If you give the command

C 1380,0

The Debugger returns 516D.

The C command uses the STCR and LDCR instructions. For more information on these instructions, see Sections 9 and 24.3.2.

The corresponding CRU output bits can be altered by inputting data in the same format that it is given by the Debugger. If a change is followed by <return>, control is returned to the Debugger. If a change is followed by <space>, the CRU input bits are displayed again.

Reading CRU data immediately after changing CRU data at the same address does not always give the same value because the CRU input and output may not have the same hardware configuration at the same address.

For example, the Disk Drive Controller CRU address >1100 is designed to enable the disk Device Service Routine ROM. This is a write-only CRU bit, so there is no input circuit. Performing an SBO instruction on address >1100 enables the Device Service Routine ROM. However, reading CRU output data from address >1100 gives meaningless data.

In addition, in some devices, such as the clock, bits may be altered by the time the CRU bit is read after writing to the CRU. Thus, the same data cannot be read even though you write the data correctly.

23.5 EXECUTE--E

Syntax definition:

E[<address>]<return>

Example:

E 2000<return> Enters the program with the parameters defined by the R command, starting at address >2000.

Definition:

Enters the program with the parameters defined by the R command, starting at the address provided. If the optional address is entered, execution begins at that address rather than at the address specified by the R command. Normally, you should use the Q command rather than the E command since the Q command restores the screen and updates the screen offset, VDP address, and screen width.

23.6 FIND WORD OR BYTE--F

Syntax definition:

F<start address>{<space> or <,>}<stop address>{<space> or <,>}<data>
{<return> or <->}

Example:

F 2000 3003 AF10<return>	Compares the data in addresses >2000 through >3003 with >AF10 and displays the addresses and contents of those that match.

Definition:

Compares the values in the start address through the stop address with the values given by the data and displays the addresses and contents of those that match. If the final entry is <return>, words are compared. If the final entry is <->, bytes are compared. If words are compared, the address is incremented by 2 before the next comparison so that comparisons are made on word boundaries. If a byte comparison is performed, the address is only incremented by one after each comparison.

A G or V can be entered at the end of the start address to indicate that the comparison should be made at the address in GROM or VDP, respectively, rather than in CPU memory. The F command is the opposite of the K command.

23.7 GROM BASE CHANGE--G

Syntax definition:

G

Example:

G — Causes the current GROM Read Data address to be displayed. It can then be altered.

Definition:

Displays the present GROM Read Data address. This address can be altered by typing the new address desired and pressing <return>. This procedure allows you to use the commands that access GROMs or GRAMs to be used in the GROM library memory areas or any other GROM address spaces. The read and write addresses are >1000 apart. The default address is >9800.

Note: This command is not useful unless you are developing special hardware.

For example, if you enter G, the response is

GROM BASE = 9800

You can then enter a new address if the device is a GRAM or press <return> to accept the address.

23.8 INSPECT SCREEN LOCATION--I

Syntax definition:

```
I
[<new screen address>]<return>
[<new screen offset>]<return>
[<new width>]<return>
```

Example:

```
I
VDP SCREEN ADDRESS = 0000 <return>
SCREEN OFFSET = 0000 60<return>
WIDTH (>20 or 28) = 0020 <return>
```

Displays the previous beginning screen address and accepts it. Displays the previous screen offset and changes it to >0060. Displays the previous width and accepts it.

Definition:

Displays the current beginning screen address, screen offset, and width, and allows you to change them. The screen address (Screen Image Table) normally starts at VDP address >0000. To move this table, you inform the Debugger with the I command. Note that the Debugger does not change the VDP Registers. It sets up a temporary word that contains the new address. The Registers are changed the next time the Debugger is entered via a breakpoint. If the Debugger is entered in another way, such as by LINK (from TI BASIC) or the LOAD AND RUN option (from the Editor/Assembler), the changes are not made.

The screen offset is useful if you are using TI BASIC because the ASCII values of the TI BASIC screen characters are offset by >60.

If your program is running in text mode, you can set the screen width to >28 rather than the default of >20.

The following example shows the use of the I command to change the screen address to >0400, the screen offset to >60, and the width to >28.

```
I
VDP SCREEN ADDRESS = 0000 400<return>
SCREEN OFFSET      = 0000 60<return>
WIDTH (>20 or 28)  = 0020 28<return>
```

23.9 FIND DATA NOT EQUAL--K

Syntax definition:

K<start address>{<space> or <,>}<stop address>{<space> or <,>}<data>
{<return> or <->}

Example:

K 2000 2100 AF10<return>	Compares the data in addresses >2000 through >2100 with >AF10 and displays the addresses and contents of those that do not match.

Definition:

Compares the values in the start address through the stop address with the values given by the data and displays the addresses and contents of those that do not match. If the final entry is <return>, words are compared. If the final entry is <->, bytes are compared. If words are compared, the address is incremented by two before the next comparison so that comparisons are made on word boundaries. If a byte comparison is performed, the address is only incremented by one after each comparison.

A G or V can be entered at the end of the start address to indicate that the comparison should be made at the address in GROM or VDP, respectively, rather than in CPU memory.

23.10 MEMORY INSPECT/CHANGE--M

Syntax definitions:

- To see or alter a memory location:

  M<address><return>
  [<data>]{<return> or <space> or <->}

- To see multiple memory locations:

  M<start address>{<space> or <,>}<stop address><return>

Examples:

M 2000<return> 2000=1000 A35C<return>	Changes the values in address >2000 from >1000 to >A35C.
M 2000 2100<return>	Displays addresses >2000 through >2100 and their values.

Definition:

Displays the address and its value if a single address is given. If a start address and stop address are given, displays the addresses and their values.

If a single address is given, followed by <return> and a value, that value is placed in the address. If a change is followed by <return>, control is returned to the Debugger. If a change is followed by <space>, the next address and its value are displayed and you can change that value. If a change is followed by <->, the previous address is displayed and you can change it. Note that entering a change followed by <space> <-> allows you to check the memory location that you have just changed.

The following example shows how to change the value in address >4042 from >1234 to >5678 and then check the change. The example assumes that address >4044 contains >CDEF. The final <return> returns you to the Debugger.

```
M 4042<return>
4042 = 1234 5678<space>
4044 = CDEF <->
4042 = 5678 <return>
```

If you choose to see multiple memory locations, each line of the display consists of the address of the first memory location followed by 12 bytes of data and the ASCII representation of the data with an asterisk {) for the unprintable characters. The display process can be halted by pressing <escape>. Press any key to stop a list of memory locations temporarily and press a key again to resume the list.

A G or V can be entered at the end of the start address to indicate that the comparison should be made at the address in GROM or VDP, respectively, rather than in CPU memory. When GROM or VDP memory are being addressed, data is accessed and altered one byte at a time. Only addresses >0000 through >3FFF of VDP RAM can be accessed. GROM cannot be altered.

Note: Even accessing GROM can alter the GROM program counter, preventing correct return to your program.

To change a VDP register, enter an address of V8 followed by the register number (0 through 7), the data byte, and <return>. For example, the following loads VDP Register 1 with >60.

```
M V8160<return>
```

Note: You cannot read VDP Registers; you can only write to them.

23.11 MOVE BLOCK--N

Syntax definition:

N<from address>{<space> or <,>}<to address>{<space> or <,>}<byte count>
{<return> or <->}

Example:

N 2000V C000 100<return>	Moves the 100 bytes starting at address >2000 in VDP RAM to the 100 bytes starting at address >C000 in CPU RAM.

Definition:

Moves the number of bytes specified. If the command is terminated with <return>, the bytes are transferred in the following order.

From Byte	To Byte
from address	to address
from address + 1	to address + 1
from address + 2	to address + 2
.	.
.	.
from address + byte count - 1	to address + byte count - 1

If the command is terminated with <->, the bytes transfer in the following order.

From Byte	To Byte
from address + byte count - 1	to address + byte count - 1
from address + byte count - 2	to address + byte count - 2
from address + byte count - 3	to address + byte count - 3
.	.
.	.
from address	to address

With the first method, you can effectively copy the same byte into several consecutive memory locations. For instance, if >13 is in address >2000, the following command places >13 in addresses >2000 through >2100.

N 2000 2001 100<return>

23.12 COMPARE MEMORY BLOCKS--P

Syntax definition:

P<start address 1>{<space> or <,>}<start address 2>{ <space> or <,>}
<byte count><return>

Example:

P 2000 3000G 100<return>	Compares the 100 bytes starting at address >2000 in CPU RAM to the 100 bytes starting at address >3000 in the GROM and prints the addresses whose data do not match and what is actually in those addresses.

Definition:

Compares the number of bytes given in byte count from the locations starting at start address 1 to the locations starting at start address 2 and prints the addresses whose data do not match and what is actually in those addresses. A G or V can be entered at the end of start address 1 or start address 2 to indicate that the comparison should be made with the address in GROM or VDP, respectively, rather than in CPU memory.

23.13 QUIT DEBUGGER--Q

Syntax definition:

Q<return>

Example:

Q<return>	Leaves the Debugger and executes the program whose parameters were defined by the R command.

Definition:

Leaves the Debugger, restores the screen, updates data entered by the I command, and executes the program whose parameters were defined by the R command. If the Program Counter is equal to >0000, the Debugger returns to the Editor/Assembler screen. You can then run your program with either the LOAD AND RUN or RUN option.

23.14 INSPECT OR CHANGE WP, PC, AND SR--R

Syntax definition:

R<return>
[<data>]{<return> or <space>}
[<data>]{<return> or <space>}
[<data>]<return>

Example:

R<return>	Shows the Workspace Pointer value and allows you to change it.

Definition:

Shows the Workspace Pointer, Program Counter, and Status Register and allows changes to be made in their values. After you open a register, you can change it. Then press <return> to return to the Debugger or <space> to open the next register for inspection or change.

The Workspace Pointer points to the program workspace. The Program Counter points to the first instruction of the program to be executed. The Status Register contains the status of the program. These values are passed to the program when the Debugger commands Q, E, or S are executed.

23.15 EXECUTE IN STEP MODE--S

Note: Executing a program one step at a time requires special hardware that is not available for the TI Home Computer. Without this special hardware, the S command has the same effect as the E command.

Syntax definition:

S[<step count>]<return>

Example:

S<return> Executes one step in the program and returns control to the Debugger.

Definition:

With the special hardware required, enters the program with the parameters defined by the R command, executes step count steps, shows the values in the Workspace Pointer, Program Counter, and Status Register, and returns to the Debugger. If step count is omitted, the default is 1.

Caution should be used when single-stepping through a section of your program that sets up VDP write addresses because the Debugger also changes VDP write addresses. You should also avoid single-stepping through code which accesses the GROM because of possible alterations to the GROM program counter.

23.16 TRADE SCREEN--T

Syntax definition:

T

Example:

T — Trades the Debugger screen for the screen as it was when the program stopped at a breakpoint.

Definition:

Trades the Debugger screen for the screen as it was when the program stopped at a breakpoint. You remain in the Debugger and can continue to use the Debugger commands. However, using commands again scrolls the screen.

23.17 TOGGLE OFFSET TO AND FROM TI BASIC--U

Syntax definition:

U

Example:

U Changes the offset by plus or minus >60.

Definition:

Changes the screen offset by plus or minus >60 to allow you to alternate displaying the screen as TI BASIC uses it and as the Debugger uses it. The ASCII characters used by TI BASIC are offset >60 from the way in which the Debugger uses those characters.

23.18 VDP BASE CHANGE--V

Syntax definition:

V<address><return>
[<data>]<return>

Example:

V<return>	Causes the present VDP Read Data address to be displayed. The address can be altered by typing a new address and pressing <return>.

Definition:

Causes the present VDP Read Data address to be displayed. The address can be altered by typing a new address and pressing <return>. This procedure allows you to use the commands that access VDP to be used in the VDP library memory areas or any other VDP address space. The read and write addresses are >1000 apart. The default address is >8800. **Note:** At present, no alternate VDP memory spaces exist.

For example, if you enter V, the response is

VDP BASE = 8800

You can then enter a new address or press <return> to accept this address.

23.19 INSPECT OR CHANGE REGISTERS--W

Syntax definition:

W[<register number>]<return>
[<data>][<return> or <space> or <->]
...

Example:

W <return> Displays all of your Workspace Registers and their values.

Definition:

Displays all of your Workspace Registers and their values if no Workspace Register number or data are given and the command is followed by <return>. If a Workspace Register number is given, it and its value are displayed and the value can be changed. After changing a value, you can press <return> to return to the Debugger, press <space> to display the next Workspace Register and its value so that you can alter that value, or press <-> to display the previous Workspace Register and its value so that you can alter that value.

23.20 CHANGE BIAS--X, Y, OR Z

Syntax definition:

- Change X bias:

  X<value><return>

- Change Y bias:

  Y<value><return>

- Change Z bias:

  Z<value><return>

Example:

Y 34<return> Changes the Y bias to >34.

Definition:

Changes the X, Y, or Z bias to the value given. After a value is assigned, you can give the characters X, Y, and Z following an address in any other command and alter that address by the amount of the bias. If the result of the alteration is an odd value, the actual address displayed is the next lower value.

For example, if the Y bias has been set to >34, the following M command displays the value in address >0134.

M 100Y<return>

23.21 HEXADECIMAL TO DECIMAL CONVERSION-->

Syntax definition:

><hexadecimal value>{<space> or <return>}

Example:

> 34<space> Displays the decimal value of >34, which is 52.

Definition:

Displays the decimal value of up to four hexadecimal digits. Values of >8000 to >FFFF are interpreted as two's-complement numbers and are thus given as negative decimal numbers.

23.22 DECIMAL TO HEXADECIMAL CONVERSION--.

Syntax definition:

.<decimal value>{<space> or <return>}

Example:

. 52<space> Displays the hexadecimal value of 52, which is >34.

Definition:

Displays the hexadecimal value of any decimal value from -32768 through 65535. For negative numbers the sign precedes the number. Values less than 0 are returned in two's-complement form. Thus, both the value 65535 and -32768 are returned as >FFFF.

23.23 HEXADECIMAL ARITHMETIC--H

Syntax definition:

H<first number>{<space> or <,>}<second number><return>

Example:

H A 6<return> Displays the sum, difference, product, quotient, and remainder of >A and >6.

Definition:

Displays the sum, difference, product, quotient, and remainder of the first number and second number. Each hexadecimal number can have up to four digits. In the example, the information is displayed as follows.

```
H1=000A  H2=0006  H1+H2 = 0010
H1-H2 = 0004  H1*H2 = 0000 003C
H1/H2 = 0001  R 0004
```

SECTION 24: APPENDICES

The following are the appendices contained in this section.

24.1 NUMBERING SYSTEMS AND ORGANIZATION

The following sections discuss the decimal, binary, and hexadecimal number systems, followed by a description of the byte and word organization in the TI Home Computer and the two's-complement representation of negative numbers.

24.1.1 Binary Number System

The number system we commonly use, using the digits 0 through 9, is known as the decimal system, or base 10. In a decimal number, each place value represents a power of 10. For example, the number 1111 in the decimal system represents the following.

$$1111_{10} = (1 \times 10^3) + (1 \times 10^2) + (1 \times 10^1) + (1 \times 10^0)$$

The binary (base 2) number system uses only the two digits 0 and 1. In a computer, these digits represent two electronic states, off and on. Each place of the binary number represents a power of two. For example, the number 1111 in the binary system is interpreted as follows.

$$1111_2 = (1 \times 2^3) + (1 \times 2^2) + (1 \times 2^1) + (1 \times 2^0) = 15_{10}$$

The decimal numbers 0 through 15 can each be represented by a four-digit binary number as shown in the following table. Each four-digit binary number corresponds to a one-digit hexadecimal (base 16) number, represented by the digits 0 through 9 and the upper-case letters A through F.

Decimal Number	Binary Number	Hexadecimal Number
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

24.1.1.1 Hexadecimal Notation

The hexadecimal (base 16) numbering system is often used as a convenient shorthand method for representing binary numbers. As the previous table shows, any four-digit binary number can be represented by one hexadecimal digit. In this manual (and in Assembler source statements), hexadecimal numbers are preceded by a greater-than sign (>).

The following illustrates the relationship between a binary number and its hexadecimal equivalent.

$0010\ 1010\ 0001\ 1110_2$ = >2A1E

24.1.2 Byte Organization

A bit (binary digit) is the smallest unit of computer information. It corresponds directly to the electronic circuitry of the computer. A bit is either on or off, and thus can be used to make either/or distinctions. For example, a bit can distinguish between yes or no, up or down, on or off, one or zero, or any two opposites. Bits are usually represented in the binary number system.

Four bits equal a nybble and eight bits equal a byte. A byte is the smallest addressable unit of information in the Home Computer. In the Home Computer, the most-significant (left-most) bit (MSB) is designated as bit 0 and the least significant (right-most) bit (LSB) is designated as bit 7.

Memory Byte

```
          (MSB)                   (LSB)
          ------------------------
bit       0  1  2  3  4  5  6  7
```

24.1.3 Word Organization

Two bytes, or 16 bits, of memory constitute a word. The computer can process a maximum of one word of information at a time. In the Home Computer, the most significant bit (MSB) of a word of memory is bit 0, while the least significant bit (LSB) is bit 15.

Memory Word

```
          (MSB)                                                   (LSB)
          ------------------------------------------------------------
bit       0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
```

The 16 bits of a word can represent such things as a machine language computer instruction, a memory address, the bit configurations of two characters, or a number. If the contents of a word are to be interpreted as a number, the number may be interpreted as a signed number in the range of -32,768 through +32,767 or as an unsigned number in the range of 0 through 65,535. Signed numbers are designated in two's-complement form. See Section 24.1.4

Each word begins at an even-numbered address (location) in memory. The even-address byte contains bits 0 through 7 of the word, and the odd-address byte contains bits 8 through 15. When word instructions address an odd byte, the computer automatically accesses the preceding even-numbered byte. All instructions must begin on a word boundary. Instructions are 1, 2, or 3 words long.

24.1.4 Two's Complement

In the Home Computer, negative numbers are represented in two's-complement form. In two's-complement form, the left-most bit of a computer word is designated as the sign bit, which indicates whether the number is positive or negative. The sign bit does not function as a part of the value of the number but only indicates positive or negative, 0 or 1, respectively.

The binary number to be subtracted (the subtrahend) must be "complemented" by changing all the 0's to 1's and all the 1's to 0's. Then 1 is added to the complemented binary number to get the two's complement. The following examples show how to find the two's complement of 26 and 53.

00011010	26 (>1A)	00110101	53 (>35)
11100101	complemented	11001010	complemented
+1	plus 1	+1	plus 1
11100110	2's complement (-26, >E6)	11001011	2's complement (-53, >CB)

Notice that the bit at the far left of the byte (MSB) is always 1 when a negative number is represented. Conversely, the MSB is 0 if a positive number is represented.

When performing subtraction, the computer converts the binary number to be subtracted to a negative number by using two's complement and then adds. In the previous example, 26 (00011010_2) was converted to negative 26 (11100110_2) or the two's complement. The following example demonstrates how to subtract 26 from 53.

00110101		00110101		53	>35
-00011010	becomes	+11100110	or	-26	+>E6
		100011011		27	>1B

The answer is 11011_2, or 27, with the ninth digit disappearing because in a byte arithmetic operation the computer recognizes only eight digits in one byte, leaving the correct answer of 00011011_2.

24.2 MEMORY ORGANIZATION

To understand memory organization, you must understand some basic terms and how they apply to the TI Home Computer.

The Central Processing Unit (CPU) of the computer contains all the circuitry for arithmetic functions, comparisons, hardware registers, and all other functions that actually process computer instructions. The CPU processes all commands and instructions fed into the computer and accesses all memory spaces. The CPU in the Home Computer is the TMS9900 Microprocessor.

One way to divide memory is into RAM (Random Access Memory) and ROM (Read Only Memory). RAM is a memory which can be written to, or read from, by any program. It stores programs and data. ROM is a memory which can only be read but not altered by any program. It is used to store information used by the computer itself, such as the built-in TI BASIC language and the makeup of the alphanumeric characters.

So that you can refer to any specific byte in the computer's memory, each byte is assigned a number. These sequential numbers, called the addresses of the bytes, are unique within each of the computer's memory spaces. They are usually referred to in hexadecimal notation.

The TMS9900 microprocessor has an address space of 64K bytes. In the Home Computer, some of this address space contains RAM and some contains ROM. In addition, some addresses are used for access to special devices, such as sound and speech, and to other areas of memory, such as VDP RAM and GROMs.

24.2.1 Directly Addressable Memory

When all possible devices are connected, 64K (65,536 or >10000) bytes of memory are directly addressable.

Addresses >0000 through >1FFF are built into the console. They contain 8K bytes of ROM that contain the TI BASIC language and other information necessary to the functioning of the computer.

Addresses >2000 through >3FFF are the 8K bytes of RAM that make up the low memory of the Memory Expansion unit. They can only be used when the Memory Expansion unit is connected.

Addresses >4000 through >5FFF are built into various peripherals. They contain up to 8K bytes of ROM for the Device Service Routine used to run peripheral devices, such as disk drives and printers. These ROMs are selected by CRU operations (see Section 9), so several ROMs can be at the same address.

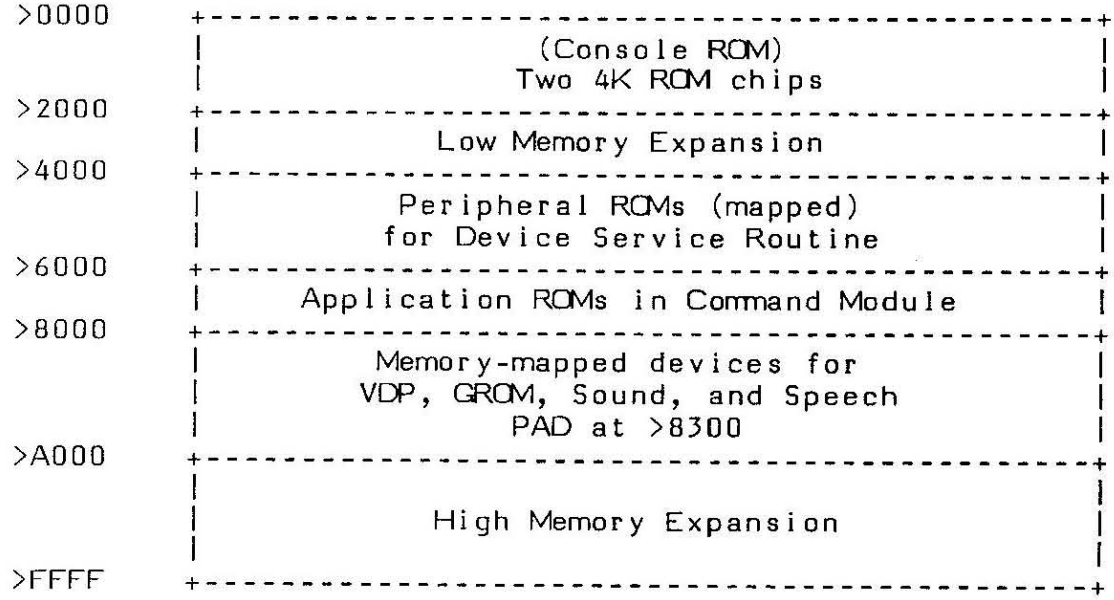
Addresses >6000 through >7FFF are available on the Command Module port. Some Command Modules, for example TI Extended BASIC, have ROM in this space.

Addresses >8000 through >9FFF are built into the console. They contain PAD from addresses >8300 through >83FF (see Section 24.3.1) and all of the memory-mapped device locations.

Addresses >A000 through >FFFF are the 24K bytes of RAM that make up the high memory of the Memory Expansion unit. They can only be used when the Memory Expansion unit is connected.

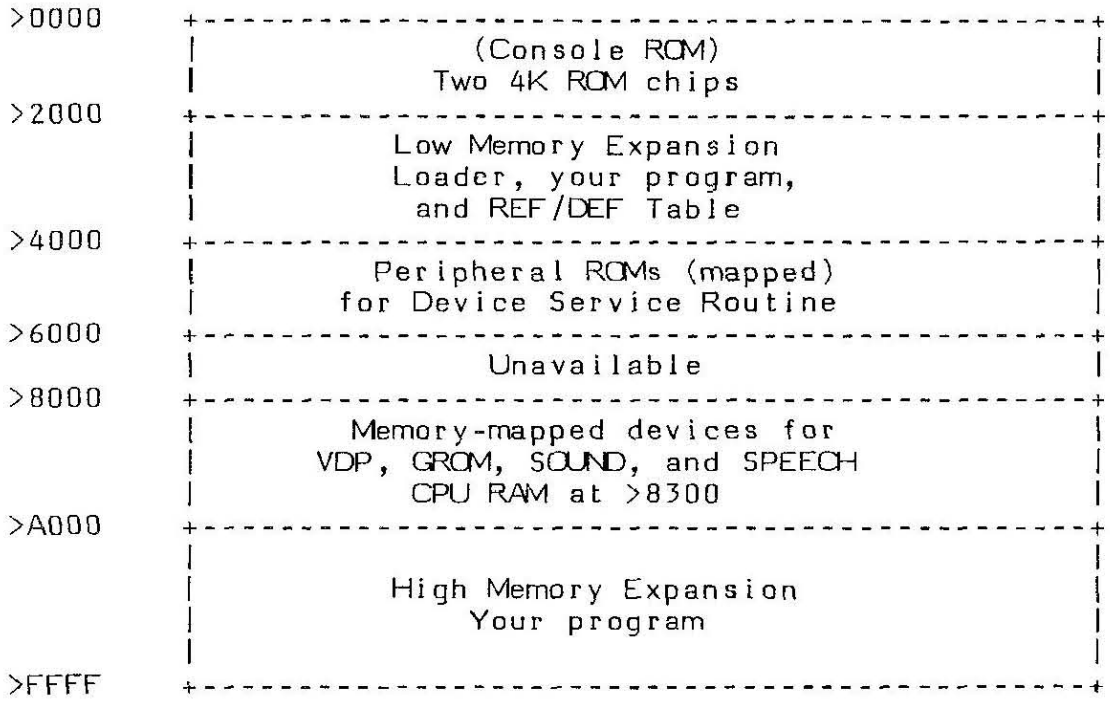
The following memory map summarizes the above information.

CPU Memory Use
General Case



When the LOAD AND RUN option of the Editor/Assembler is in use, the following diagram shows how the memory is used.

CPU Memory Use
LOAD AND RUN Option from the Editor/Assembler



24.2.1.1 Expansion RAM

The Memory Expansion unit is a 32K-byte peripheral on an eight-bit bus. It has two blocks of memory, an 8K block from >2000 through >3FFF and a 24K block from addresses >A000 through >FFFF. Addresses >FFD8 through >FFFF are used for XOP 1 on the TI-99/4A. When your program is executing, the 24K block contains your program and the 8K block contains the Loader, your program (if it was too large to fit in the 24K block), utilities, and the REF/DEF table.

24.2.1.2 ROM

All the ROMs (Read Only Memory) are directly accessible by an assembly language program. Two 4K-byte console ROMs are located at addresses >0000 through >1FFF. They contain the operating system, the GPL interpreter, and part of the TI BASIC interpreter.

The memory block at addresses >4000 through >5FFF is assigned to the peripheral ROMs which can be accessed by setting the bit assigned for the CRU (Communication Register Unit) to enable the particular ROM. These ROMs contain DSRs (Device Service Routines), and they are located in a peripheral. See Section 9 for more information.

Application ROMs are contained in command modules. They occupy address >6000 through >7FFF.

24.2.1.3 GROM

A GROM (Graphics Read Only Memory) is another type of ROM. It is designed to contain GPL (Graphic Programming Language) programs which are executed by the GPL interpreter in the console. GPL is commonly used in applications software and can only be executed through a GROM.

A GROM is a memory-mapped device, just as VDP RAM is. A GROM's memory is addressed by writing its address to a specific CPU address and reading data from another specified CPU address. See Section 16.5 for a discussion of accessing GROMs.

GROM addresses are from >0000 through >F7FF. Each GROM has 6K bytes of memory that start from an even-numbered first-digit address. For example, GROM 0 is at addresses >0000 through >17FF and GROM 1 is at addresses >2000 through >37FF. The computer can access up to eight GROMs at a time.

GROMs 0, 1, and 2 are in the console and contain the monitor program, part of the operating system, and most of the TI BASIC interpreter. Five additional GROMS can be located in a Command Module. The number of GROMs used in a Command Module depends on the size of the applications program.

24.2.2 Memory-Mapped Devices

The memory-mapped devices are VDP (Video Data Processing) RAM, the Speech Synthesizer, the sound processor, GROMs, and so forth. VDP RAM is discussed in this section. For discussions of other memory-mapped devices, see Sections 16, 20, 21, and 22.

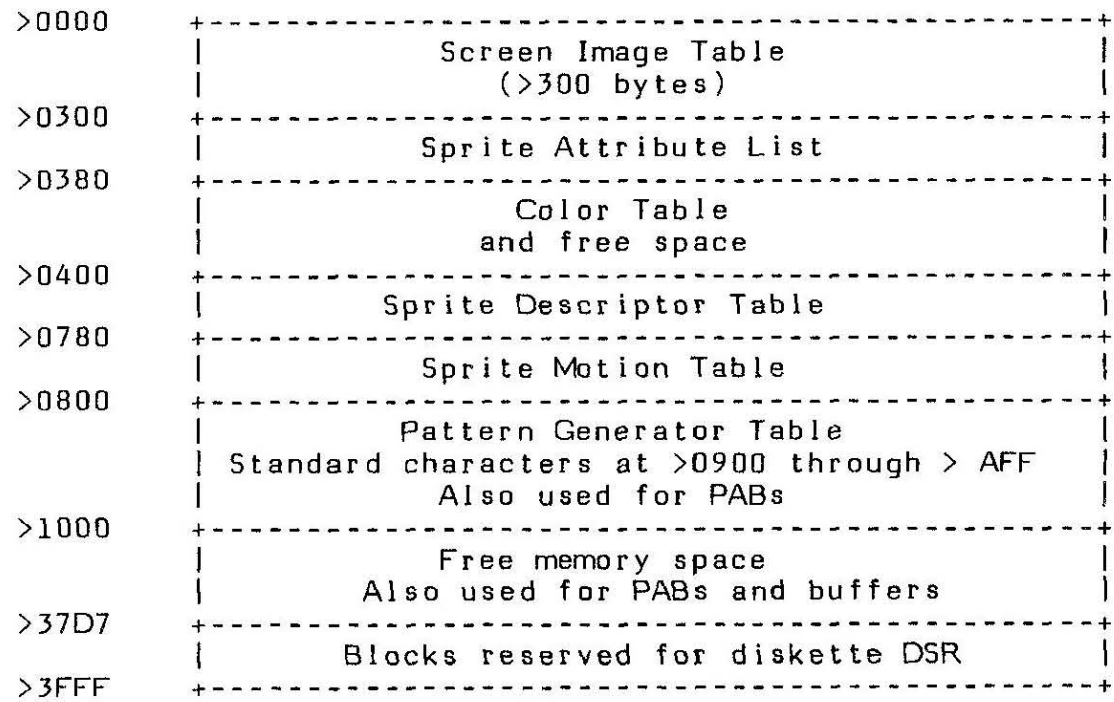
The Video Display Processor (VDP) RAM, located in the console, is used chiefly for common video functions, such as screen images, character pattern tables, color tables, etc. See Section 21 for a discussion of the use of VDP RAM for screen-related functions.

When TI BASIC is in use, VDP RAM also contains the TI BASIC program, the program symbol table, the value stack, and the string space. VDP RAM is also used as a storage space by applications programs. Part of VDP RAM is used as a data buffer. Another part of VDP RAM functions as a PAB (Peripheral Access Block) to pass information from a file to the appropriate DSR (Device Service Routine). Assembly language programs cannot be executed from VDP RAM.

VDP RAM is a memory-mapped area of 16K (16,384 or >4000) bytes numbered >0000 through >3FFF. VDP RAM addresses are automatically incremented, so only one address in CPU RAM is required to read or write a specific block of data. Assigned addresses exist for each I/O function in the RAM. For example, the VDP RAM read data Register is located at CPU RAM address >8800, the VDP read status Register is found at CPU RAM address >8802, the VDP write data Register is at CPU RAM address >8C00, and the VDP write address Register is at CPU RAM address >8C02. See Section 16 for more information on writing to and reading from VDP RAM.

The diagram on the next page shows the memory of VDP RAM when it is being used by the Editor/Assembler.

VDP RAM Memory Use
Editor/Assembler



24.3 MEMORY, CRU, AND INTERRUPT STRUCTURE

The following gives the structure and addresses of the memory use in PAD at addresses >8300 through >83FF, CRU use, and interrupt handling.

24.3.1 CPU RAM PAD Use

CPU RAM PAD is located at addresses >8300 through >83FF. The system software uses the high memory locations at addresses >83C0 through >83FF. The rest of the CPU RAM is used according to the type of routines being executed. The following describes the use of this memory.

Addresses	Use
>8300 - >8349	Available for use by your program, with some limitations. The system software uses this area only for temporary storage. However, if your assembly language program is executed through a TI BASIC program by CALL LOAD and CALL LINK, and the assembly language program returns control to TI BASIC, only the area from >8300 through >8317 is available. Further, if parameters are passed by CALL LINK, then only the area from >8300 through >830F is available. However, if TI BASIC is used only to load and transfer control to an assembly language program, all of this area is available to your program.
>834A - >836D	Used as a stack area by the interpreter, floating point routines, and DSR routines. Unless console routines are called by your assembly language program, this area is available for use.
>836E - >836F	Available for your use. However, if the TI BASIC interpreter or floating point routines are running, this area is used as a value stack pointer.
>8370 - >837F	Used for the GPL status block as follows.
>8370	Contains the highest available address of VDP RAM.
>8372	The least-significant byte of the data stack pointer. The most-significant byte is >83. When the computer is initialized, this contains a value of >CF. However, after the first time it is accessed, it is changed to >D0.
>8373	The least-significant byte of the subroutine stack pointer. The most-significant byte is >83. When the computer is initialized, this contains a value of >7E. However, after the first time it is accessed, it is changed to >80.
>8374	The keyboard number to be scanned, with a default value of >0.

Addresses	Use
>8375	The ASCII key code detected by the scan routine.
>8376	The Wired Remote Controller Y-location.
>8377	The Wired Remote Controller X-location.
>8378	The random number generator.
>8379	The VDP interrupt timer. It is incremented every sixtieth of a second.
>837A	The number of sprites that can be in motion. It is originally set to >00.
>837B	The VDP status byte. It is a copy of the VDP status Register. Bit 0 is the 60 Hz VDP interrupt. It is on every time the screen is updated and off when the bit is read. Bit 1 is on any time there are five or more sprites on a line. Bit 2 is on any time that sprite coincidence occurs. Bits 3 through 7 contain the number of the fifth sprite on a line when there are five or more sprites on a line.
>837C	The GPL STATUS byte. All the values are controlled by the GPL interpreter. Bit 0 is the high bit. Bit 1 is the greater than bit. Bit 2 is the condition bit. The key scan routine turns this bit on when a new key is detected. Also, the DSR routine turns this bit on to indicate that a file does not exist. Bit 3 is the carry bit. Bit 4 is the overflow bit.
>837D	The character buffer used by the VDP. It reflects the code loaded in the screen image area of the VDP RAM. Loading a character code at this address results in displaying the character on the screen based on the pointers at address >837E and >837F.
>837E	Points to the current row on the screen.
>837F	Points to the current column on the screen.
>8380 - >83BF	The default subroutine stack address is >8380 and the default data stack address is >83A0. Your assembly language program may use this area unless it uses the GPLLNK routine. GPL uses subroutine stacks and data stacks while executing the routine. Thus, it is important to leave this area untouched. The TI BASIC interpreter uses address >838A through >83BF as the subroutine and data stack area. Additionally, addresses >8388 and >8389 are reserved for the TI BASIC interpreter.
>83C0 - >83DF	Interpreter Workspace. You must not use this area. The bytes are used by the interpreter as follows.

Addresses	Use
>83C0	*Random number seed.*
>83C2	TI-99/4 only: Eight bytes for remote handset debounce and internal flags. TI-99/4A only: Used as a flag byte to control interrupt routine.
>83C4	TI-99/4A only: Address of the user-defined interrupt routine.
>83CA	Console keyboard debounce.
>83CC	Sound list pointer.
>83CE	Number of the sound byte, decremented on each VDP interrupt.
>83D0	Search pointers for GROM and ROM search. Four bytes.
>83D4	Current value stored in VDP Register 1.
>83D6	Screen time-out counter, decremented on each VDP interrupt. Screen blanks after the word reaches >0000. Upon new key detection, the word is reset.
>83D8	Return address saved by the scan routine.
>83DA	Player number used by the scan routine.
>83E0 - >83FF	GPL Workspace Registers. This area is used as a register area by all the console routines, including the GPL and TI BASIC interpreters. Use of the registers depends on the routine being executed. However, Registers 13, 14, and 15 always contain the GROM write address, the system flags, and the VDP write address, respectively.

24.3.2 CRU Allocation

The Communication Register Unit (CRU) is used for system access to peripherals. There are 4K CRU bits, numbered >0000 through >0FFF. The CRU address loaded into Workspace Register 12 is twice the bit number. Thus, loading Workspace Register 12 with >1000 sets the base equal to CRU bit >800. (See Section 9 for more information.) Of the available 4K of CRU bits, the first K, at addresses >0000 through >07FE, are used internally by the console. This includes the TMS9901 I/O chip, which addresses the keyboard, joysticks, cassette, etc. See Section 24.3.3 for more information on I/O mapping.

The second K, at addresses >0800 through >0FFE, are reserved for future use.

The last 2K, at addresses >1000 through >1FFE, are reserved for the peripherals that are attached to the console port. A block of 128 CRU bits is assigned to each peripheral as shown below. A0 through A15 are the CPU address bus lines.

CRU Addresses	A3	A4	A5	A6	A7	Use (Peripheral)	Device Number
>0000 - >0FFE	0	x	x	x	x	Internal use	
>1000 - >10FE	1	0	0	0	0	Reserved	0
>1100 - >11FE	1	0	0	0	1	Disk controller	1
>1200 - >12FE	1	0	0	1	0	Reserved	2
>1300 - >13FE	1	0	0	1	1	RS232, ports 1 and 2	3
>1400 - >14FE	1	0	1	0	0	Reserved	4
>1500 - >15FE	1	0	1	0	1	RS232, ports 3 and 4	5
>1600 - >16FE	1	0	1	1	0	Reserved	6
>1700 - >17FE	1	0	1	1	1	Reserved	7
>1800 - >18FE	1	1	0	0	0	Thermal Printer	8
>1900 - >19FE	1	1	0	0	1	Future expansion	9
>1A00 - >1AFE	1	1	0	1	0	Future expansion	10
>1B00 - >1BFE	1	1	0	1	1	Future expansion	11
>1C00 - >1CFE	1	1	1	0	0	Future expansion	12
>1D00 - >1DFE	1	1	1	0	1	Future expansion	13
>1E00 - >1EFE	1	1	1	1	0	Future expansion	14
>1F00 - >1FFE	1	1	1	1	1	P-Code peripheral	15

CRU address 0 at A8 through A14 is the memory enable bit in each device address space. Setting the bit to 1 turns the device ROM/RAM on, and resetting it to 0 turns it off. This enables the address space from >4000 through >5FFF reserved for the peripheral ROM.

24.3.3 Interrupt Handling

The highest priority interrupts are the reset and load vectors with a priority of 0. The reset interrupt is used when the computer is turned on. Interrupt priority 1 connects through the TMS9901 Programmable Systems Interface for interrupt expansion. The following shows the interrupts available.

9900 Interrupts

Interrupt Level	Vector Memory Address	CPU Pin	Device Assignment
Highest	>0000	RESET	Reset
0	>FFEC	LOAD	Load
1	>0004	INT1	External Device (TMS9901)

Note that the lower priority CPU interrupts are not used. The following additional interrupts are implemented on the TMS9901.

9901 Interrupt Mapping

Address	CRU Bit	9901	Pin	Function
>0000	0	Control		Control.
>0002	1	INT1	17	External.
>0004	2	INT2	18	VDP vertical synchronization.
>0006	3	INT3	9	Clock interrupt, keyboard enter line, and joystick fire button.
>0008	4	INT4	8	Keyboard 1 line and joystick left.
>000A	5	INT5	7	Keyboard p line and joystick right.
>000C	6	INT6	6	Keyboard 0 line and joystick down.
>000E	7	INT7 (P15)	34	Keyboard shift line and joystick up.
>0010	8	INT8 (P14)	33	Keyboard space line.
>0012	9	INT9 (P13)	32	Keyboard q line.
>0014	10	INT10 (P12)	31	Keyboard 1 line.
>0016	11	INT11 (P11)	30	Not used.
>0018	12	INT12 (P10)	29	Reserved.
>001A - >001E	13 - 15	INT13 - INT15	28, 27, 23	Not used.
>0020	16	P0	38	Reserved.
>0022	17	P1	37	Reserved.
>0024	18	P2	26	Bit 2 of keyboard select.
>0026	19	P3	22	Bit 1 of keyboard select.

Address	CRU Bit	9901	Pin	Function
>0028	20	P4	21	Bit 0 of keyboard select.
>002A	21	P5	20	Alpha lock on the TI-99/4A.
>002C	22	P6	19	Cassette control 1.
>002E	23	P7 (INT15)	23	Cassette control 2.
>0030	24	P8 (INT14)	27	Audio gate.
>0032	25	P10 (INT12)	28	Magnetic tape output.
>0036	27	P11 (INT11)	30	Magnetic tape input.
>0038 - >003E	28 - 31	P12 - P15	31 - 34	Not used.

24.4 COMPARISONS WITH TI EXTENDED BASIC LOADER

The following sections compare the Loader that the Editor/Assembler uses with the Loader used by TI Extended BASIC. The major differences involve memory use, speed, external references, utility references, entry points, duplicate definitions, tags, and use of some routines.

24.4.1 Memory Use

One of the basic differences between the Editor/Assembler Loader and the TI Extended BASIC Loader is the use of the memory in the Memory Expansion unit.

The TI Extended BASIC interpreter uses the high memory location starting at address >A000 for its own program space and data. Its utilities are loaded at the low memory area starting at address >2000. The Loader only recognizes the area between addresses >2000 and >3FFF (size about 8K) as the area for an assembly language program.

The Editor/Assembler Loader, however, recognizes both areas (addresses >2000 through >3FFF and >A000 through >FFD7), and the Loader is loaded at address >2000. The Loader checks the high memory area first. If there is not enough space left to load the program it is loaded into the low memory area. Thus, the Editor/Assembler Loader has more space (about 32K including utility routines and the Loader itself) for programs than the TI Extended BASIC Loader.

When using TI Extended BASIC, you can load the program in the high memory area by loading absolute code with a program which starts with an AORG statement. However, extreme caution must then be taken if the TI Extended BASIC program needs to run after the load because the TI Extended BASIC program code, line number table, and numeric values are located in the high memory starting from the top of the memory.

The easiest way to find out the available memory space in the high memory area is to do a SIZE command after running the TI Extended BASIC program once. The program space displayed on the screen is the space available for the assembly language program. Note that this value is given in decimal notation.

The other way to find the highest memory address available for assembly language programs is to execute the CALL PEEK statement at address >8386. For example, execute CALL PEEK(-31866,A,B) and print A and B. This address contains the

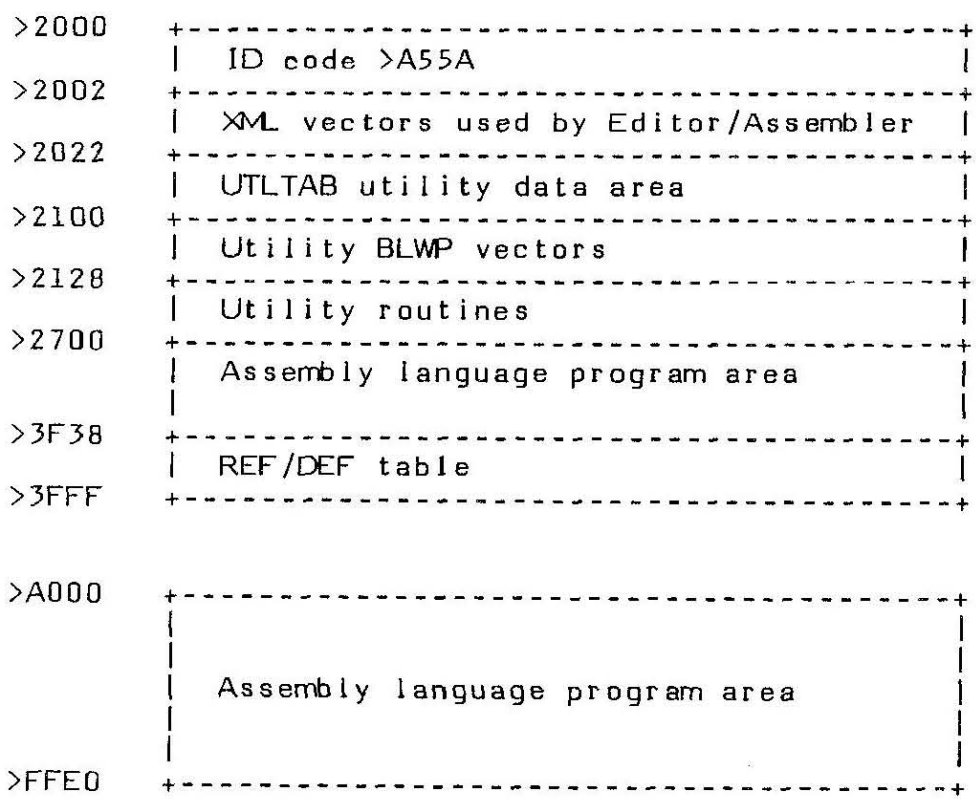
pointer to the highest free address in the expansion memory. Convert this value to a hexadecimal value, and compare it with the last address in the assembly language program.

It is possible for your assembly language program and the TI Extended BASIC program to overwrite each other if one or both of the programs use large amounts of space.

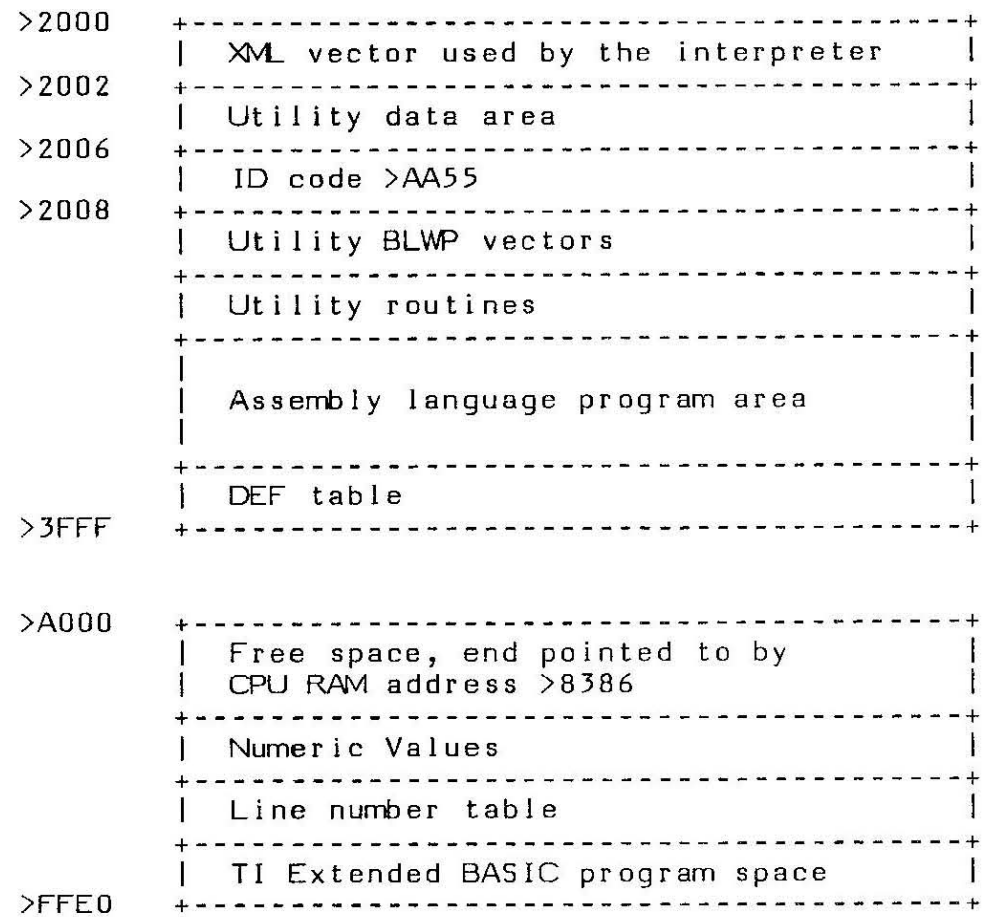
The utility routines include a Name Link Routine to search the program and other utility routines provided by the Loader. The basic memory configuration of both Loaders is similar, except that the Editor/Assembler Loader has more utility routines available and has free higher memory spaces.

The memory usage of the two Loaders is shown in the following figures.

Memory Expansion Unit Use
by the Editor/Assembler Loader



Memory Expansion Unit Use
by the TI Extended BASIC Loader



24.4.2 Loading Speed

Both Loaders are tagged object loaders which load a fixed 80 display format file from a diskette. The difference between the Loaders is the greater speed of the Editor/Assembler Loader. Also, it handles compressed object files, which the TI Extended BASIC Loader cannot handle.

24.4.3 External References

The Editor/Assembler Loader is a linking loader which handles external references. Thus, a program can be broken into different files, and any section of the program can be referred to by the other files by means of REF and DEF instructions. The TI Extended BASIC Loader does not allow external references.

For example, if both the following programs are loaded and run, the Editor/Assembler resolves all references.

```
            DEF     DATA1,PRGM
*
DATA1       EQU     >1234
*
PRGM        ...                 Beginning of the program.
            .
            .
            .
            END


            DEF     MAIN
            REF     DATA1,PRGM,KSCAN
*
MAIN        ...                 Beginning of the program.
            .
            .
            .
            LI      R1,DATA1
            .
            .
            .
            BL      @PRGM
            .
            .
            .
            END
```

24.4.4 Utility References

The TI Extended BASIC Loader only handles DEF statements and DEFed labels that are entered by the Loader as it loads the program. In order to access the utility routines in TI Extended BASIC, equated addresses must be specified in the program by EQU instructions. See Section 24.4.8 for descriptions of the equated addresses.

The Editor/Assembler Loader creates pre-defined labels of all the utility routines and frequently accessed memory addresses. Thus, REF instructions in your program are sufficient to access the utility routines, and object tags 3 and 4 are accepted by this Loader.

24.4.5 Entry Point

Another feature of the Editor/Assembler Loader is that it allows a program entry point to be specified with the END instructions. Any address label with the END statement is considered to be the address and executes the program immediately without comming back to the Editor/Assembler screen. Thus, object tags 1 and 2 are accepted by this Loader.

24.4.6 Duplicate Definition

Duplicate definition is allowed by the TI Extended BASIC Loader, with the Loader replacing the new defintion with the old definition. Thus, if two programs with the same entry name are loaded, the most recently loaded program is executed when the name is specified.

The Editor/Assembler Loader, however, issues a DUPLICATE DEFINITION error message, and loading stops.

24.4.7 Tags

The comparison of the object tags is listed on the next page.

Tags	Function	Extended BASIC Loader	Editor/Assembler Loader
0	Module ID	Supported	Supported
1, 2	Entry address	Ignored	Supported
3, 4	External REFs	Issues error	Supported
5, 6	External DEFs	Supported	Supported
7, 8	Checksum	Supported	Supported
9, A	Load address	Supported	Supported
B, C	Data	Supported	Supported
D, E	Load bias	Issues error	Issues error
F	End of record	Supported	Supported
G, H	Unused	Issues error	Issues error
I	Program ID	Ignored	Ignored
M	Data/common seg.	Ignored	Issues error
Other		Issues error	Issues error

In order to access the utility routines of the TI Extended BASIC Loader, all the utility references must be done by equating (with EQU) to the routine address.

24.4.8 TI Extended BASIC Equates

The following shows the equates used in TI Extended BASIC.

```
VDPWA   EQU   >8C02
VDPWD   EQU   >8C00
VDPRD   EQU   >8800
VDPSTA  EQU   >8802
FAC     EQU   >834A
GPLWS   EQU   >83E0
PAD     EQU   >8300
SOUND   EQU   >8400
SPCHRD  EQU   >9000
SPCHWT  EQU   >9400
GRMRD   EQU   >9800
GRMRD   EQU   >9802
GRMWD   EQU   >9C00
GRMWA   EQU   >9C02
SCAN    EQU   >000E
*
* Utility Branches
```

```
*
NUMASG  EQU     >2008
NUMREF  EQU     >2000
STRASG  EQU     >2010
STRREF  EQU     >2014
XMLLNK  EQU     >2018
KSCAN   EQU     >201C
VSBW    EQU     >2020
VMBW    EQU     >2024
VSBR    EQU     >2028
VMBR    EQU     >202C
VWTR    EQU     >2030
ERR     EQU     >2034
FADD    EQU     >0D80
FSUB    EQU     >0D7C
FMUL    EQU     >0E88
FDIV    EQU     >0FF4
SADD    EQU     >0D84
SSUB    EQU     >0D74
SMUL    EQU     >0E8C
SDIV    EQU     >0FF8
CSN     EQU     >11AE
CFI     EQU     >12B8
FCOMP   EQU     >0D3A
NEXT    EQU     >0070
COMPCT  EQU     >00
GETSTR  EQU     >02
MEMCHK  EQU     >04
CNS     EQU     >06
VPUSH   EQU     >0E
VPOP    EQU     >10
ASSGNV  EQU     >18
CIF     EQU     >20
SCROLL  EQU     >26
VGWITE  EQU     >34
GVWITE  EQU     >36
```

```
*
* Error Equates
*
ERRNO   EQU   >0200      2  Numeric Overflow.
ERRSYN  EQU   >0300      3  Syntax Error.
ERRIBS  EQU   >0400      4  Illegal After Subprogram.
ERRNQS  EQU   >0500      5  Unmatched Quotes.
ERRNTL  EQU   >0600      6  Name Too Long.
ERRSNM  EQU   >0700      7  String-Number Mismatch.
ERROBE  EQU   >0800      8  Option Base Error.
ERRMUV  EQU   >0900      9  Improperly Used Name.
ERRIM   EQU   >0A00     10  Image Error.
ERRMEM  EQU   >0B00     11  Memory Full.
ERRSO   EQU   >0C00     12  Stack Overflow.
ERRNWF  EQU   >0D00     13  NEXT Without FOR.
ERRFNN  EQU   >0E00     14  FOR-NEXT Nesting.
ERRSNS  EQU   >0F00     15  Must Be in Subprogram.
ERRRSC  EQU   >1000     16  Recursive Subprogram Call.
ERRMS   EQU   >1100     17  Missing SUBEND.
ERRRWG  EQU   >1200     18  RETURN Without GOSUB.
ERRST   EQU   >1300     19  String Truncated.
ERRRBS  EQU   >1400     20  Bad Subscript.
ERRSSL  EQU   >1500     21  Speech String Too Long.
ERRLNF  EQU   >1600     22  Line Not Found.
ERRBLN  EQU   >1700     23  Bad Line Number.
ERRLTL  EQU   >1800     24  Line Too Long.
ERRCC   EQU   >1900     25  Can't Continue.
ERRCIP  EQU   >1A00     26  Command Illegal in Program.
ERROLP  EQU   >1B00     27  Only Legal in a Program.
ERRBA   EQU   >1C00     28  Bad Argument.
ERRNPP  EQU   >1D00     29  No Program Present.
ERRBV   EQU   >1E00     30  Bad Value.
ERRIAL  EQU   >1F00     31  Incorrect Argument List.
ERRINP  EQU   >2000     32  Input Error.
ERRDAT  EQU   >2100     33  Data Error.
ERRFE   EQU   >2200     34  File Error.
ERRIO   EQU   >2400     36  I/O Error.
ERRSNF  EQU   >2500     37  Subprogram Not Found.
ERRPV   EQU   >2700     39  Protection Violation.
ERRIVN  EQU   >2800     40  Unrecognized Character.
WRNNO   EQU   >2900     41  Numeric Overflow.
```

```
WRNST   EQU   >2A00     42  String Truncated.
WRNNPP  EQU   >2B00     43  No Program Present.
WRNINP  EQU   >2C00     44  Input Error.
WRNIO   EQU   >2D00     45  I/O Error.
```

24.4.9 Subprogram Use

All the TI BASIC interface and support routines in the TI Extended BASIC Loader are supported by the Editor/Assembler Loader. However, some routines work slightly differently. The use of these subprograms with TI BASIC when the Editor/Assembler module is attached is described in Section 17. The following describe how they differ from the TI Extended BASIC routines with the same names.

CALL INIT--Loads the utility routines from a TI Extended BASIC program or from the Editor/Assembler command module. This subprogram functions similarly, but the code that accomplishes it is different.

CALL LOAD--The Editor/Assembler is forced to load the utility routines if they have not been loaded by INIT routine. This is not done in the TI Extended BASIC Loader, and an an error is issued if INIT has not been called.

CALL LINK--The Editor/Assembler Loader uses its own workspace to store information from the parameter list. Only the addresses >8310 through >8312 are reserved for parameter passing purposes. TI Extended BASIC uses addresses >8300 through >8315 for this information. Your assembly language program must not modify this area if parameters are to be accessed in the program.

CALL PEEK--The Editor/Assembler allows the TI BASIC program to peek more than one consecutive memory area in a statement by means of a null string delimiter, whereas TI Extended BASIC only allows one consecutive memory area to be peeked in each statement.

CALL PEEKV, CALL POKEV--These are not supported by TI Extended BASIC.

ERROR LINK ROUTINE--The error code used with the TI Extended BASIC support routine may not give the same error message when the program is run in the Editor/Assembler environment, due to the fact that the TI BASIC and TI Extended BASIC interpreters have different error messages and error handling routines.

Note also that the error messages given by the Editor/Assembler while executing TI BASIC interface routines use the console TI BASIC error messages. They are very often different from the error message issued by TI Extended BASIC. For example, the STRING-NUMBER MISMATCH error issured by TI Extended BASIC is a SYNTAX ERROR when issued by the Editor/Assembler since the TI BASIC in the console does not contain the STRING-NUMBER MISMATCH error message.

The TI Extended BASIC support routines which are used to access numeric and string parameters are slightly modified in the Editor/Assembler version in order to utilize console routines in the console TI BASIC interpreter. However, assembly language programs access them exactly the same way as with the TI Extended BASIC utilities.

24.5 SAVE UTILITY

The SAVE utility alows you to save TMS9900 tagged object code in memory image format on either diskette or cassette. In this format programs can be run by using the RUN PROGRAM FILE option on the Editor/Assembler.

The SAVE program is on the Editor/Assembler diskette labeled Part B under the name SAVE. The program is executed with the LOAD AND RUN option on the Editor/Assembler. However, before it is run, you must include certain DEFs in your program, assemble it, and load it.

Your program must contain DEF SFIRST,SLAST,SLOAD, with these symbols defined as follows.

SFIRST must be a pointer to the start of your program. Further, the start of the program must be an executable instruction. However, SFIRST does not necessarily reflect where the memory image program is loaded.

SLOAD must be the address where the saved program is to be located. Since memory image programs are not relocatable, SLOAD should usually equal SFIRST.

SLAST must be a pointer to the address after the last word of your program. This can be done most easily by making SLAST the label for the END directive in your program.

Then choose the LOAD AND RUN option. For the first file name, enter the name of your object file: for example, DSK1.OBJECT. Then enter the name of the SAVE program, DSK1.SAVE. Then press <return> to go to the next prompt, and enter the program name as SAVE. The SAVE utility then executes. It displays reminders about using the utility and prompts for a file name for the file that it creates.

To output to cassette rather than diskette, enter the file name CS1.

The SAVE utility can only save 8K in one file. If your program is larger than that (SFIRST minus SLAST is greater than >2000 bytes), then the SAVE utility creates a second file by incrementing the last byte of the current file name, thus creating a new file name. The Editor/Assembler option RUN PROGRAM FILE expects files to be linked in this way.

Note: If you save a file on CS1 that is larger than 8K, do not rewind the cassette when saving the additional files, even though you are instructed to do so. Otherwise you will save the additional files over the first one. When loading programs that consist of more than one file using the RUN PROGRAM FILE option, give the input file name as CS1. If you give it as CS1, then the RUN PROGRAM FILE option will increment the name to CS2 for the second file, and CS2 is an invalid file name in RUN PROGRAM FILE.

24.6 SPEECH SYNTHESIZER RESIDENT VOCABULARY

The following is a list of all the letters, numbers, words, and phrases that can be accessed, followed by the location of their codes in ROM. See Section 22 for a description of the use of speech.

Phrase	Address	Phrase	Address
- (NEGATIVE)	48DC	+ (POSITIVE)	51B3
. (POINT)	50EC	0	13C3
1	1409	2	145C
3	149A	4	14E7
5	1531	6	15A8
7	15E8	8	1637
9	1664		
A (ay)	16E4	A1 (uh)	1700
ABOUT	1714	AFTER	1769
AGAIN	17A5	ALL	1807
AM	1830	AN	1876
AND	18AC	ANSWER	1913
ANY	1962	ARE	556E
AS	19A7	ASSUME	19E8
AT	1A25		
B	1A42	BACK	1A64
BASE	1A8F	BE	1A42
BETWEEN	1ADE	BLACK	1B47
BLUE	1B8A	BOTH	1BB6
BOTTOM	1BEA	BUT	1C20
BUY	1C48	BY	1C48
BYE	1C48		
C	1C86	CAN	1CD9
CASSETTE	1D10	CENTER	1D47
CHECK	1D82	CHOICE	1DA2
CLEAR	1DE6	COLOR	1E20
COME	1E54	COMES	1E87
COMMA	1EDE	COMMAND	1F1A
COMPLETE	1F71	COMPLETED	1FCD
COMPUTER	2034	CONNECTED	208B

Phrase	Address	Phrase	Address
CONSOLE	20F3	CORRECT	213C
COURSE	2182	CYAN	21C0
D	2203	DATA	223C
DECIDE	2294	DEVICE	22FD
DID	2366	DIFFERENT	23C4
DISKETTE	242D	DO	2480
DOES	24B3	DOING	24EA
DONE	253E	DOUBLE	2599
DOWN	25D3	DRAW	2612
DRAWING	2668		
E	26CB	EACH	26F0
EIGHT	1637	EIGHTY	2723
ELEVEN	2759	ELSE	27B6
END	27F5	ENDS	2866
ENTER	28AD	ERROR	28EF
EXACTLY	2937	EYE	3793
F	299F	FIFTEEN	29C2
FIFTY	2A1D	FIGURE	2A60
FIND	2AD7	FINE	2B1E
FINISH	2B5B	FINISHED	2B94
FIRST	2BD7	FIT	2C14
FIVE	1531	FOR	14E7
FORTY	2C3E	FOUR	14E7
FOURTEEN	2C7F	FOURTH	2D19
FROM	2D74	FRONT	2DBC
G	2DEB	GAMES	2E28
GET	2E8C	GETTING	2EBA
GIVE	2F38	GIVES	2F8D
GO	2FFC	GOES	3031
GOING	3079	GOOD	30D6
GOOD WORK	30FA	GOODBYE	3148
GOT	31A0	GRAY	31D1
GREEN	321D	GUESS	327E
H	32C0	HAD	32EF
HAND	3339	HANDHELD UNIT	337F

Phrase	Address	Phrase	Address
HAS	3405	HAVE	344A
HEAD	348C	HEAR	34E5
HELLO	351A	HELP	3571
HERE	34E5	HIGHER	35AE
HIT	360A	HOME	363E
HOW	3689	HUNDRED	36EF
HURRY	3757		
I	3793	I WIN	37CF
IF	3850	IN	3872
INCH	38B5	INCHES	38FA
INSTRUCTION	394B	INSTRUCTIONS	39BD
IS	3A32	IT	3A7A
J	3AAE	JOYSTICK	3AED
JUST	3B4C		
K	3B8A	KEY	3BB9
KEYBOARD	3BE9	KNOW	3C4F
L	3C8F	LARGE	3CD0
LARGER	3D19	LARGEST	3D67
LAST	3DDE	LEARN	3E1E
LEFT	3E78	LESS	3EB2
LET	3F08	LIKE	3F2F
LIKES	3F6A	LINE	3FD5
LOAD	404B	LONG	40D3
LOOK	413D	LOOKS	4191
LOWER	41E7		
M	4233	MADE	4267
MAGENTA	42AE	MAKE	432E
ME	437D	MEAN	43BD
MEMORY	4405	MESSAGE	446C
MESSAGES	44D7	MIDDLE	4551
MIGHT	4593	MODULE	45DF
MORE	4642	MOST	4693
MOVE	46DF	MUST	473D

Phrase	Address	Phrase	Address
N	4786	NAME	47C0
NEAR	4833	NEED	4880
NEGATIVE	48DC	NEXT	4959
NICE TRY	49A5	NINE	1664
NINETY	4A4E	NO	3C4F
NOT	4AAB	NOW	4ADA
NUMBER	4B20		
O	4B7D	OF	4BBA
OFF	4C13	OH	4B7D
ON	4C41	ONE	1409
ONLY	4C8B	OR	4CDC
ORDER	4D34	OTHER	4D8A
OUT	4DD4	OVER	4E0A
P	4E66	PART	4E9F
PARTNER	4EE0	PARTS	4F31
PERIOD	4F81	PLAY	4FE5
PLAYS	502D	PLEASE	5093
POINT	50EC	POSITION	5148
POSITIVE	51B3	PRESS	5231
PRINT	526D	PRINTER	52AA
PROBLEM	52F9	PROBLEMS	5360
PROGRAM	53EE	PUT	5477
PUTTING	54AA		
Q	5520		
R	556E	RANDOMLY	55A0
READ (reed)	5652	READ1 (red)	57C1
READY TO START	56B3	RECORDER	5745
RED	57C1	REFER	5801
REMEMBER	5861	RETURN	58CF
REWIND	593A	RIGHT	7C38
ROUND	59C2		
S	5A5A	SAID	5AA1
SAVE	5AEF	SAY	5B65
SAYS	5BA2	SCREEN	5BFB
SECOND	5C5B	SEE	1C86

Phrase	Address	Phrase	Address
SEES	5CBF	SET	5D1B
SEVEN	15E8	SEVENTY	5D50
SHAPE	5DA5	SHAPES	5DDE
SHIFT	5E27	SHORT	5E5C
SHORTER	5EA5	SHOULD	5F24
SIDE	5F6D	SIDES	5FC8
SIX	15A8	SIXTY	601A
SMALL	6070	SMALLER	60AE
SMALLEST	60F1	SO	6153
SOME	6197	SORRY	61C6
SPACE	6226	SPACES	625D
SPELL	62CC	SQUARE	6333
START	637C	STEP	63C5
STOP	63F7	SUM	6197
SUPPOSED	6423	SUPPOSED TO	6489
SURE	64F4		
T	6551	TAKE	658B
TEEN	65BF	TELL	6603
TEN	664E	TEXAS INSTRUMENTS	6696
THAN	675B	THAT	67B6
THAT IS INCORRECT	6816	THAT IS RIGHT	68FE
THE (thee)	6974	THE1 (thuh)	69B6
THEIR	6A72	THEN	69E1
THERE	6A72	THESE	6ADE
THEY	6B47	THING	6BA0
THINGS	6C0F	THINK	6C73
THIRD	6CBC	THIRTEEN	6D11
THIRTY	6DA2	THIS	6DDE
THREE	149A	THREW	6E26
THROUGH	6E26	TIME	6E69
TO	145C	TOGETHER	6EB0
TONE	6F1F	TOO	145C
TOP	6F8D	TRY	6FBB
TRY AGAIN	700F	TURN	7092
TWELVE	70CE	TWENTY	7119
TWO	145C	TYPE	7170

Phrase	Address	Phrase	Address
U	71BE	UHOH	71F4
UNDER	7245	UNDERSTAND	729D
UNTIL	732F	UP	739F
UPPER	73C3	USE	7403
V	7449	VARY	7487
VERY	74DA		
W	7520	WAIT	759D
WANT	75DF	WANTS	7621
WAY	76B0	WE	767D
WEIGH	76B0	WEIGHT	759D
WELL	7717	WERE	775C
WHAT	77BC	WHAT WAS THAT	77E9
WHEN	7875	WHERE	78AB
WHICH	78F4	WHITE	7924
WHO	7969	WHY	79B4
WILL	7A11	WITH	7A6B
WON	1409	WORD	7AAB
WORDS	7B0A	WORK	7B75
WORKING	7BBC	WRITE	7C38
X	7C8D		
Y	7CB2	YELLOW	7CF8
YES	7D58	YET	7D99
YOU	71BE	YOU WIN	7DDB
YOUR	7E4D		
Z	7E99	ZERO	13C3

24.7 CHARACTER SET

The Editor/Assembler recognizes the ASCII characters listed in the following table. The table includes the ASCII code for each character represented as both a hexadecimal and decimal value. The Editor/Assembler also recognizes the six special characters shown in the second table. On the TI-99/4A Home Computer, the Editor/Assembler also represents the lower-case letters, {, }, and the tilde as shown in the third table.

Editor/Assembler Primary Character Set

Hexadecimal Value	Decimal Value	Character
20	32	Space
21	33	!
22	34	"
23	35	#
24	36	$
25	37	%
26	38	&
27	39	'
28	40	(
29	41	)
2A	42	*
2B	43	+
2C	44	,
2D	45	-
2E	46	.
2F	47	/
30	48	0
31	49	1
32	50	2
33	51	3
34	52	4
35	53	5
36	54	6
37	55	7
38	56	8
39	57	9
3A	58	:

Hexadecimal Value	Decimal Value	Character
3B	59	;
3C	60	<
3D	61	=
3E	62	>
3F	63	?
40	64	@
41	65	A
42	66	B
43	67	C
44	68	D
45	69	E
46	70	F
47	71	G
48	72	H
49	73	I
4A	74	J
4B	75	K
4C	76	L
4D	77	M
4E	78	N
4F	79	O
50	80	P
51	81	Q
52	82	R
53	83	S
54	84	T
55	85	U
56	86	V
57	87	W
58	88	X
59	89	Y
5A	90	Z

Editor/Assembler Special Characters

Hexadecimal Value	Decimal Value	Character
5B	91	[
5C	92	\
5D	93	]
5E	94	^
5F	95	_
60	96	`

TI-99/4A Editor/Assembler Additional Characters

Hexadecimal Value	Decimal Value	Character
61	97	a
62	98	b
63	99	c
64	100	d
65	101	e
66	102	f
67	103	g
68	104	h
69	105	i
6A	106	j
6B	107	k
6C	108	l
6D	109	m
6E	110	n
6F	111	o
70	112	p
71	113	q
72	114	r
73	115	s
74	116	t
75	117	u
76	118	v
77	119	w
78	120	x
79	121	y

Hexadecimal Value	Decimal Value	Character
7A	122	z
7B	123	{
7D	125	}
7E	126	~

24.8 ASSEMBLER DIRECTIVE TABLE

The assembler directives for the TI Home Computer are listed in the following table. All directives may include a comment field following the operand field. Those directives that do not require an operand field may have a comment field following the operator field. Those directives that have optional operand fields (RORG and END) may have comment fields only when they have operand fields.

The following symbols and conventions are used in defining the syntax of assembler directives.

Angle brackets (< >) enclose items you supply.
Brackets ([ ]) enclose optional items.
An ellipsis (...) indicates that the preceding item may be repeated.
Braces ({ }) enclose two or more items of which one must be chosen.

The following words are used in defining the items used in assembler directives.

label	A symbol used in the label field.
string	A character string of a length defined for each directive.
expr	An expression.
wd expr	A well-defined expression.
term	The term used to refer to an extended operation.
operation	A mnemonic operation code, macro name, or previously defined operation or extended operation.

Assembler Directives

Directive	Syntax	Force Word Boundary	Note
Page TiTLe	[<label>] TITL <string>	NA	
Program IDTntifier	[<label>] IDT <string>	NA	
External DEFinition	[<label>] DEF <symbol>[,<symbol>]...	NA	
External REFerence	[<label>] REF <symbol>[,<symbol>]...	NA	
COPY file	[<label>] COPY "<file name>"	NA	
Absolute ORiGin	[<label>] AORG <wd expr>	No	
Relocatable ORiGin	[<label>] RORG [<expr>]	No	2
Dummy ORiGin	[<label>] DORG [<expr>]	No	

Directive	Syntax	Force Word Boundary	Note
Block Starting with Symbol	[<label>] BSS <wd expr>	No	
Block Ending with Symbol	[<label>] BES <wd expr>	No	
Initialize word	[<label>] DATA <expr>[,<expr>]...	Yes	
Initialize TEXT	[<label>] TEXT [-] <string>	No	1
Define eXtended OPeration	[<label>] DXOP <symbol>, <term>	NA	
Define assembly-time constant	[<label>] EQU <expr>	NA	2
Word boundary	[<label>] EVEN	Yes	
No source List	[<label>] UNL	NA	
LIST Source	[<label>] LIST	NA	
PAGE eject	[<label>] PAGE	NA	
Initialize BYTE	[<label>] BYTE <wd expr>[,<wd expr>]...	No	
Program END	[<label>] END [<symbol>]	NA	3
Program SEGment	[<label>] PSEG	Yes	4
Program segment END	[<label>] PEND	Yes	4
Data SEGment	[<label>] DSEG	Yes	4
Data segment END	[<label>] DEND	Yes	4
Common SEGment	[<label>] CSEG [<string>]	Yes	4
Common segment END	[<label>] CEND	Yes	4
Secondary REFerence	[<label>] SREF <symbol>[,<symbol>]...	NA	4
Force LOAD	[<label>] LOAD <symbol>[,<symbol>]...	NA	4

Notes:

[1]The minus sign causes the Assembler to negate the right-most character.
[2]Symbols in expressions must have been previously defined.
[3]Symbol must have been previously defined.
[4]These directives have no effect when using the Loader provided with the Editor/Assembler.

24.9 HEXADECIMAL INSTRUCTION TABLE

The following table lists the TMS9900 assembly language instructions, their format, and the section in which they are described. They are in order according to their hexadecimal operation code. For an alphabetical listing by their mnemomic operation code, see Section 24.10. See Section 5 for an explanation of the format.

Hexadecimal Operation Code	Mnemonic Operation Code	Name	Format	Section
0200	LI	Load Immediate	VIII	10.1
0220	AI	Add Immediate	VIII	6.4
0240	ANDI	AND Immediate	VIII	11.1
0260	ORI	OR Immediate	VIII	11.2
0280	CI	Compare Immediate	VIII	8.3
02A0	STWP	STore Workspace Pointer	VIII	10.7
02C0	STST	STore STatus	VIII	10.6
02E0	LWPI	Load Workspace Pointer Immediate	VIII	10.3
0300	LIMI	Load Interrupt Mask Immediate	VIII	10.2
0340	IDLE	IDLE	VII	9.6
0360	RSET	ReSET	VII	9.6
0380	RTWP	ReTurn with Workspace Pointer	VII	7.17
03A0	CKON	ClocK ON	VII	9.6
03C0	CKOF	ClocK OFf	VII	9.6
03E0	LREX	Load or REstart eXecution	VII	9.6
0400	BLWP	Branch And Load Workspace Pointer	VI	7.3
0440	B	Branch	VI	7.1
0480	X	EXecute	VI	7.18
04C0	CLR	CLeaR operand	VI	11.5
0500	NEG	NEGate	VI	6.11
0540	INV	INVert	VI	11.4
0580	INC	INCrement	VI	6.8
05C0	INCT	INCrement by Two	VI	6.9
0600	DEC	DECrement	VI	6.5
0640	DECT	DECrement by Two	VI	6.6
0680	BL	Branch and Link	VI	7.2

Hexadecimal Operation Code	Mnemonic Operation Code	Name	Format	Section
06C0	SWPB	SWaP Bytes	VI	10.8
0700	SETO	SET to One	VI	11.6
0740	ABS	ABSolute value	VI	6.3
0800	SRA	Shift Right Arithmetic	V	12.1
0900	SRL	Shift Right Logical	V	12.2
0A00	SLA	Shift Left Arithmetic	V	12.3
0B00	SRC	Shift Right Circular	V	12.4
1000	JMP	Unconditional JuMP	II	7.11
1100	JLT	Jump Less Than	II	7.10
1200	JLE	Jump if Low Or Equal	II	7.9
1300	JEQ	Jump EQual	II	7.4
1400	JHE	Jump High Or Equal	II	7.6
1500	JGT	Jump Greater Than	II	7.5
1600	JNE	Jump Not Equal	II	7.13
1700	JNC	Jump No Carry	II	7.12
1800	JOC	Jump On Carry	II	7.16
1900	JNO	Jump No Overflow	II	7.14
1A00	JL	Jump if logical Low	II	7.8
1B00	JH	Jump if logical High	II	7.7
1C00	JOP	Jump Odd Parity	II	7.15
1D00	SBO	Set CRU Bit to One	II	9.2
1E00	SBZ	Set CRU Bit to Zero	II	9.3
1F00	TB	Test Bit	II	9.5
2000	COC	Compare Ones Corresponding	III	8.4
2400	CZC	Compare Zeros Corresponding	III	8.5
2800	XOR	EXclusive OR	III	11.3
2C00	XOP	EXtended OPeration	IX	7.19
3000	LDCR	LoaD CRU	IV	9.1
3400	STCR	STore CRU	IV	9.4
3800	MPY	MultiPlY	IX	6.10
3C00	DIV	DIVide	IX	6.7
4000	SZC	Set Zeros Corresponding	I	11.9
5000	SZCB	Set Zeros Corresponding, Byte	I	11.10
6000	S	Subtract words	I	6.12
7000	SB	Subtract Bytes	I	6.13
8000	C	Compare words	I	8.1
9000	CB	Compare Bytes	I	8.2

Hexadecimal Operation Code	Mnemonic Operation Code	Name	Format	Section
A000	A	Add words	I	6.1
B000	AB	Add Bytes	I	6.2
C000	MOV	MOVe words	I	10.4
D000	MOVB	MOVe Bytes	I	10.5
E000	SOC	Set Ones Corresponding	I	11.7
F000	SOCB	Set Ones Corresponding, Byte	I	11.8

24.10 ALPHABETICAL INSTRUCTION TABLE

The following table lists the TMS9900 assembly language instructions, their format, and the section in which they are described. They are in alphabetical order by their mnemonic operation code. For a listing in order according to their hexadecimal operation code, see Section 24.9. See Section 5 for an explanation of the format.

Hexadecimal Operation Code	Mnemonic Operation Code	Name	Format	Section
A000	A	Add words	I	6.1
B000	AB	Add Bytes	I	6.2
0740	ABS	ABSolute value	VI	6.3
0220	AI	Add Immediate	VIII	6.4
0240	ANDI	AND Immediate	VIII	11.1
0440	B	Branch	VI	7.1
0680	BL	Branch and Link	VI	7.2
0400	BLWP	Branch And Load Workspace Pointer	VI	7.3
8000	C	Compare words	I	8.1
9000	CB	Compare Bytes	I	8.2
0280	CI	Compare Immediate	VIII	8.3
03C0	CKOF	ClocK OFf	VII	9.6
03A0	CKON	ClocK ON	VII	9.6
04C0	CLR	CLeaR operand	VI	11.5
2000	COC	Compare Ones Corresponding	III	8.4
2400	CZC	Compare Zeros Corresponding	III	8.5
0600	DEC	DECrement	VI	6.5
0640	DECT	DECrement by Two	VI	6.6
3C00	DIV	DIVide	IX	6.7
0340	IDLE	IDLE	VII	9.6
0580	INC	INCrement	VI	6.8
05C0	INCT	INCrement by Two	VI	6.9
0540	INV	INVert	VI	11.4
1300	JEQ	Jump EQual	II	7.4
1500	JGT	Jump Greater Than	II	7.5
1B00	JH	Jump if logical High	II	7.7
1400	JHE	Jump High Or Equal	II	7.6
1A00	JL	Jump if logical Low	II	7.8
1200	JLE	Jump if Low Or Equal	II	7.9

Hexadecimal Operation Code	Mnemonic Operation Code	Name	Format	Section
1100	JLT	Jump Less Than	II	7.10
1000	JMP	Unconditional JuMP	II	7.11
1700	JNC	Jump No Carry	II	7.12
1600	JNE	Jump Not Equal	II	7.13
1900	JNO	Jump No Overflow	II	7.14
1800	JOC	Jump On Carry	II	7.16
1C00	JOP	Jump Odd Parity	II	7.15
3000	LDCR	LoaD CRU	IV	9.1
0200	LI	Load Immediate	VIII	10.1
0300	LIMI	Load Interrupt Mask Immediate	VIII	10.2
03E0	LREX	Load or REstart eXecution	VII	9.6
02E0	LWPI	Load Workspace Pointer Immediate	VIII	10.3
C000	MOV	MOVe words	I	10.4
D000	MOVB	MOVe Bytes	I	10.5
3800	MPY	MultiPlY	IX	6.10
0500	NEG	NEGate	VI	6.11
0260	ORI	OR Immediate	VIII	11.2
0360	RSET	ReSET	VII	9.6
0380	RTWP	ReTurn with Workspace Pointer	VII	7.17
6000	S	Subtract words	I	6.12
7000	SB	Subtract Bytes	I	6.13
1D00	SBO	Set CRU Bit to One	II	9.2
1E00	SBZ	Set CRU Bit to Zero	II	9.3
0700	SETO	SET to One	VI	11.6
0A00	SLA	Shift Left Arithmetic	V	12.3
E000	SOC	Set Ones Corresponding	I	11.7
F000	SOCB	Set Ones Corresponding, Byte	I	11.8
0800	SRA	Shift Right Arithmetic	V	12.1
0B00	SRC	Shift Right Circular	V	12.4
0900	SRL	Shift Right Logical	V	12.2
3400	STCR	STore CRU	IV	9.4
02C0	STST	STore STatus	VIII	10.6
02A0	STWP	STore Workspace Pointer	VIII	10.7

Hexadecimal Operation Code	Mnemonic Operation Code	Name	Format	Section
06C0	SWPB	SWaP Bytes	VI	10.8
4000	SZC	Set Zeros Corresponding	I	11.9
5000	SZCB	Set Zeros Corresponding, Byte	I	11.10
1F00	TB	Test Bit	II	9.5
0480	X	EXecute	VI	7.18
2C00	XOP	EXtended OPeration	IX	7.19
2800	XOR	EXclusive OR	III	11.3

24.11 PROGRAM ORGANIZATION

You can write your program so that it returns to the Editor/Assembler, to the TI BASIC or TI Extended BASIC program that called it, or to the master title screen. The program must retain the return address and must not have altered the GPL Workspace Registers.

24.11.1 Returning When Your Program Is Run Automatically

The final instruction in your program can be an END instruction followed by a label which has been mentioned in a DEF instruction. Then the program is run automatically when it is loaded by the Editor/Assembler LOAD AND RUN option, the Editor/Assembler RUN option, or the TI BASIC or TI Extended BASIC statement CALL LOAD. When your program is run automatically, the user Workspace equate (USRWS EQU >20BA) is not loaded. To use this area for your Workspace Registers, use the following sequence.

```
USRWS   EQU   >20BA
        LWPI  USRWS
```

You may define your own Workspace Register area instead.

When your program is run automatically, it starts in the Graphics Progamming Language Workspace (starting at GPLWS). The return address is in Workspace Register 11. Do not use GPLWS as your Workspace. Instead, branch to your own area, save the return address from GPL Workspace Register 11, and set up your own Workspace. When your program is done, return by clearing the GPL STATUS byte at address >837C to indicate that there are no errors, put the return address in your Workspace Register 11, and return. The following program segment shows these steps.

```
SAVRTN  DATA  0
MYWS    BSS   20
STATUS  EQU   >837C
        .
        .
        .
```

```
START   MOV     R11,@SAVRTN     Save GPL return address.
        LWPI    MYWS            Set up Workspace Registers.
        .
        .
        .
        CLR     R0              Prepare to return to GPL.
        MOVB    R0,@STATUS      Indicate no errors.
        MOV     @SAVRTN,R11     Load return address.
        RT                      Return.
        END     START
```

24.11.2 Returning When Your Program Is Not Run Automatically

If you run your program by answering the PROGRAM NAME prompt in the Editor/Assembler LOAD AND RUN or RUN options or with the CALL LINK statement in TI BASIC, the user Workspace equate (USRWS EQU >20BA) is loaded and you can use the area that starts at that address for your Workspace Registers or you can define your own Workspace Register area instead.

The address in Workspace Register 11 does not need to be saved unless you use the Workspace Register for some other purpose. When you wish to return, simply clear the GPL STATUS byte and return. The following program segment shows these steps.

```
        DEF     START
STATUS  EQU     >837C
START   .
        .
        .
        CLR     R0              Prepare to return to GPL.
        MOVB    R0,@STATUS      Indicate no errors.
        RT                      Return.
        END
```

24.11.3 Other Returns

You can return to the calling program by branching to address >0070. Before doing this, clear the GPL STATUS byte to indicate that there are no errors and load the GPL Workspace Registers. The following program segment shows these steps.

```
CLR     R0
MOVB    R0,@STATUS      Indicate no errors.
LWPI    GPLWS           Load GPL Workspace Registers.
B       @>0070          Return.
```

You can return to the master title screen by enabling interrupts, loading the GPL Workspace Registers, and branching through the vector >0000. The following program segment shows these steps.

```
LIMI    2               Enable interrupts.
LWPI    GPLWS           Load GPL Workspace Registers.
BLWP    @>0000          Return to color bar screen.
```

24.12 ERROR MESSAGES

The following sections give the error messages that may be returned by the Editor/Assembler.

24.12.1 Input/Output Error Codes

The following table lists the input/output error codes.

Error Code	Meaning
0	Bad device name.
1	Device is write protected.
2	Bad open attribute such as incorrect file type, incorrect record length, incorrect I/O mode, or no records in a relative record file.
3	Illegal operation;, i.e., an operation not supported on the peripheral or a conflict with the OPEN attributes.
4	Out of table or buffer space on the device.
5	Attempt to read past the end of file. When this error occurs, the file is closed. Also given for non-existing records in a relative record file.
6	Device error. Covers all hard device errors such as parity and bad medium errors.
7	File error such as program/data file mismatch, non-existing file opened in INPUT mode, etc.

24.12.2 Error Messages Issued by GROM Code

The following are the error messages that may be issued by code in a GROM.

NAME TOO LONG
NO MEMORY EXPANSION
I/O ERROR n, where n is the I/O error code from 0 through 7.

24.12.3 Errors Issued by the Loader

The following are the error messages that may be issued by the Loader.

I/O ERROR n, where n is the I/O error code from 0 through 7.
MEMORY FULL
ILLEGAL TAG
CHECKSUM ERROR
DUPLICATE DEFINITION
UNRESOLVED REFERENCE

24.12.4 Execution-Time Errors

The following are the error messages that may be issued at execution time.

I/O ERROR n, where n is the I/O error code from 0 through 7.
PROGRAM NOT FOUND
ERROR CODE n, where n is the error code listed below.

The table on the following page lists the errors that may be issued when you attempt to run your program.

Error Code	Meaning
00 - 07	Input/Output error
08	MEMORY FULL
09	INCORRECT STATEMENT
0A	ILLEGAL TAG
0B	CHECKSUM ERROR
0C	DUPLICATE DEFINITION
0D	UNRESOLVED REFERENCE
0E	INCORRECT STATEMENT
0F	PROGRAM NOT FOUND
10	INCORRECT STATEMENT
11	BAD NAME
12	CAN'T CONTINUE
13	BAD VALUE
14	NUMBER TOO BIG
15	STRING-NUMBER MISMATCH
16	BAD ARGUMENT
17	BAD SUBSCRIPT
18	NAME CONFLICT
19	CAN'T DO THAT
1A	BAD LINE NUMBER
1B	FOR-NEXT ERROR
1C	I/O ERROR
1D	FILE ERROR
1E	INPUT ERROR
1F	DATA ERROR
20	LINE TOO LONG
21	MEMORY FULL
22 - FF	UNKNOWN ERROR CODE

GLOSSARY

Addresses: The numbering system which defines the memory locations within the computer.

Addressing mode: A way of using memory addressing. In the Editor/Assembler, the addressing modes are Workspace Register addressing, Workspace Register indirect addressing, Workspace Register indirect auto-increment addressing, symbolic memory addressing, and indexed memory addressing.

Arithmetic greater than bit: A bit in the Status Register that is set when a signed number is compared with a smaller signed number.

Arithmetic operators: The arithmetic operators are + for addition, - for subtraction, * for multiplication, and / for signed division.

ASCII: American Standard Code for Information Interchange. The code used to represent data.

Assembler: The portion of the Editor/Assembler on Diskette A that allows you to assemble an assembly language program into object code (machine language).

Assembling: Changing an assembly language program into a machine language program, called object code, that can be run by the computer.

Assembly language: A lower-level language that allows fast access to all machine resources, including functions not available from higher-level languages.

Assembly options: Options that you provide at the time of assembly. They are R for automatic Workspace Register generation, L for list file generation, S for a symbol table, and C for compressed object format.

Assembly-time constant: An expression in the operand field of an EQU directive.

Binary: The base 2 numbering system used by the computer.

Bit I/O instructions: Format II instructions whose operand field contains a well-defined expression which evaluates to a CRU bit address, relative to the contents of Workspace Register 12.

Bit-map: A way of handling graphics in the TI-99/4A Home Computer.

Bit: A BInary digiT.

Byte: Eight bits.

Carry bit: A Status Register bit that is set by a carry of 1 from the most significant bit (sign bit) of a word or byte during arithmetic and shift operations.

Character constant: A string of one or two characters enclosed in single quotes.

Character set: The characters that are recognized by the Assembler/Linker. It is listed in Section 24.7.

Character string: A string of characters enclosed in single quotes.

Color Table: A table in memory that defines the colors of graphics.

Command mode: The mode in the Editor in which you may perform special functions, such as copying lines, deleting lines, altering data, and the like.

Comment field: An area in which to make comments that increase the readability of the program but that do not affect the operations of the computer.

Compressed object code: Object code that takes up less space on diskettes. The code has compressed hexadecimal numbers for the tagged fields.

Console: The main physical unit of the computer.

Constant: An unchanging value. The four types of constants recognized by the Assembler are decimal integer constants, hexadecimal integer constants, character constants, and assembly-time constants.

Context switch: A change in the location of the next address to be accessed by the computer.

Control and CRU instructions: The control and CRU instructions are Clock Off (CKOF), Clock On (CKON), Load CRU (LDCR), Idle (IDLE), Load or Restart Execution (LREX), Reset (RSET), Set CRU Bit to One (SBO), Set CRU Bit to Zero (SBZ), Store CRU (STCR), and Test Bit (TB).

Control instructions: Format VII instructions which require no operand field.

CPU: Central Processing Unit.

CPU RAM: Central Processing Unit Random Access Memory. Used in this manual to describe any memory that can be directly addressed by the CPU.

CRU: Communications Register Unit. A command-driven bit-addressable I/O interface.

Debugger: A program to help you check memory locations, registers, and the like in order to find and correct any errors which may occur in your program.

Decimal integer constant: A decimal number from -32,768 to +65,535. Positive decimal integer constants greater than 32,767 are considered negative when interpreted as two's complement values.

DEF/REF Table: A list of the variables which have been referred to in a DEF or REF statement in a program or series of programs.

Destination operand: The address where the result of the performed manipulation is stored.

Device Service Routine: A routine to handle communications between the computer and all external devices, such as printers, disk drives, the RS232 Interface, etc.

Directives: Instructions to the Assembler that control the assembly process. Directives affect the Location Counter and the Assembler output, initialize constants, provide linkage between programs, and have other functions.

DSR: Device Service Routine.

Edit mode: The mode in the Editor in which you may create and alter files.

Editor: The portion of the Editor/Assembler that allows you to create, edit, print, and save files.

End-of-file marker: (*EOF). The mark that indicates the end of a file that you are editing.

Equal bit: A Status Register bit that is set when the two words or bytes being compared are equal.

Expressions: Used in the operand fields of assembler directives and machine instructions. An expression is a constant or symbol, a series of constants or symbols, or a series of constants and symbols separated by arithmetic operators.

Extended operation bit: A Status Register bit that is set when an extended operation (available in some TI-99/4A Home Computers) is being executed.

Fatal error: An error which stops the assembly process.

Field: A division of a record. A source record consists of the label field, op-code field, operand field(s), and comment field.

File: A group of program statements, object code, data, or the like, contained in the computer's memory or on an external device such as a diskette.

GPL: Graphics Programming Language. The language frequently used to program Command Modules.

GROM: Graphics Read Only Memory.

Hexadecimal integer constant: A string of up to four hexadecimal numerals preceded by a greater than (>) sign. Hexadecimal numerals include the decimal values 0 through 9 and the letters A through F.

Hexadecimal: Base 16 numbering system. Often used as an easy representation of the binary numbering system.

Immediate instructions: Format VIII instructions which contain a Workspace Register address followed by a comma and an expression. Use the contents of the word following the instruction word as the operand of the instruction.

Indexed memory addressing: Specifies the memory address that contains the operand. An indexed memory address is preceded by an "at" sign (@) and followed by a register name enclosed in parentheses.

Instruction formats: One of nine formats that specify the way in which instructions are assembled to machine language.

Interrupt mask bits: Bits 12 through 15 in the Status Register. They determine what devices are permitted to interrupt the processor.

Jump instructions: Format II instructions that use Program Counter relative addresses coded as expressions corresponding to instruction locations on word boundaries.

Label field: The first field in a source record. It serves as a reference point.

List file: A file which the Assembler can create. It contains a record of the assembly process.

Loader: An assembly language program used to load assembly language programs into the Memory Expansion unit.

Loading: The process of putting object code into the computer's memory so that it can be run.

Location counter: A counter that keeps track of where the Assembler is in the assembly process.

Logical greater than bit: A Status Register bit that is set when an unsigned number is compared with a smaller unsigned number.

Machine language: The code into which assembly language is translated by the Assembler. The code produced can be recognized and operated on by the TMS9900 microprocessor.

Memory: The storage locations or addresses in the computer.

Mnemonic codes: Codes which help you to remember the instructions in assembly language.

Mode of operation: The way in which a file may be accessed. May be INPUT, OUTPUT, UPDATE, or APPEND.

Non-fatal error: An error which does not stop the assembly process.

Nybble: Four bits; half a byte.

Object code: *The machine code into which assembly language is translated by the* Assembler.

Odd parity bit: A Status Register bit that is set when the parity of the result is odd and is reset when the parity is even.

Op-code field: The second field in a source record. It is the operation code (a number, name, or abbreviation) of the task to be performed by that source statement.

Operand field: The field that stipulates the value to be operated upon or manipulated by the op-code.

Operands: The numbers, expressions, or characters upon which assembly language instructions operate.

Overflow bit: A Status Register bit that is set when the result of an arithmetic operation is too large or too small to be represented in two's complement representation.

PAB: Peripheral Access Block.

Peripheral Access Block: A set of locations in VDP memory that defines how devices, such as printers and disk drives, are accessed.

Predefined symbols: Symbols for addresses that are predefined in the DEF/REF table.

Program Counter Register: Keeps track of the location of the next instruction in memory.

Program Counter relative addressing: Used only by jump instructions. It is written as an expression that corresponds to an address at a word boundary.

Pseudo-instruction: An assembly language statement that has the form of an instruction, but is defined in terms of other instructions. The pseudo-instructions are No Operation (NOP) and Return (RT).

RAM: Random Access Memory.

Re-entrant programming: A technique that allows the same program code to be used for several different applications while maintaining the integrity of the data used with each application.

Register: A memory word that serves a secific purpose. Registers in Random Access Memory (RAM) are called "software" registers. A set of 16 consecutive registers is called a "workspace."

ROM: Read Only Memory.

Screen Image Table: A table in memory corresponding to graphics on the screen.

Source operand: The number, address, string, etc., which is to be manipulated or operated upon.

Source statements: The statements of an assembly language program.

Special keys: Special characters and functions and certain keys available for cursor movement when using the Editor/Assembler.

Sprite: One of 32 characters that may be placed on the screen and moved smoothly.

Sprite Attribute List: A list in memory that defines the location, color, and pattern of sprites.

Sprite Descriptor Table: A table in memory defining sprite patterns and sizes.

Sprite Motion Table: A table in memory that defines the motion of sprites.

STATUS byte, GPL: The byte at address >837C that contains status information.

Status Register: The register that contains indications of the present status of the computer.

Symbol table: A table constructed by the Assembler or TI BASIC in the assembly process. It lists all of the symbols used in a program and contains information on the symbols in the program, their addresses, and their types.

Symbol: A string of alphanumeric characters (A through Z and 0 through 9), the first of which must be an alphabetic character, and none of which may be a blank.

Symbolic Addresses: Addresses associated with locations in the program that must not be used in the label field of other statements.

Symbolic memory addressing: Specifies the memory address that contains the operand. A symbolic memory address is preceded by an "at" sign (@).

Syntax definition: A description of the required form for the use of commands as related to the fields.

Syntax: The required form for source statements.

T-field value: A value which indicates the type of addressing mode used.

TMS9900 microprocessor: The chip on which the TI-99/4 and TI-99/4A Home Computers are based.

Tag characters: Characters that describe the information in an object file.

Terms: A decimal or hexadecimal constant, an absolute assembly-time constant, or a label having an absolute value.

Transfer vector: Two consecutive words of memory which contain a new Workspace pointer and a new program counter. The computer uses a transfer vector to perform a transfer of control called a context switch.

Two's complement: The way in which negative numbers are expressed in binary in the computer.

Undisplayable characters: Characters that have valid ASCII meanings but cannot be displayed on the screen.

Utilities: Programs provided by Texas Instruments to enable quick and easy use of certain computer capabilities. They are VSBW, VMBW, VSBR, VMBR, VWTR, XMLLNK, KSCAN, GPLLNK, DSRLNK, SAVE, LOADER, NUMASG, STRASG, NUMREF, STRREF, and ERR.

VDP RAM: Video Display Processor Random Access Memory. This memory can be accessed indirectly.

Well-defined expressions: Expressions whose symbols or assembly-time constants have been previously defined.

Window: A 40-column area that is displayed on the screen. The entire file is 80 characters wide, made up of three overlapping windows.

Workspace Pointer Register: Contains the address of the current software workspace.

Workspace Register addressing: Specifies the Workspace Register that contains the operand.

Workspace Register indirect addressing: Specifies a Workspace Register that contains the address of the operand. An indirect Workspace Register address is preceded by an asterisk (*).

Workspace Register indirect auto-increment addressing: Specifies a Workspace Register that contains the address of the operand.

Workspace: A set of 16 consecutive words of memory.

D

E

F

J

K

L

M

N

O

P

Q

R

S

T

THREE-MONTH LIMITED WARRANTY
HOME COMPUTER SOFTWARE MEDIA

Texas Instruments Incorporated extends this consumer warranty only to the original consumer purchaser.

WARRANTY COVERAGE

This warranty covers the electronic and case components of the software program storage media. These components include all semiconductor chips and devices, diskettes, plastics, boards, wiring and all other hardware contained in this storage media ("the Hardware"). This limited warranty does not extend to the programs contained in the storage media and the accompanying book materials ("the Programs").

The Hardware is warranted against malfunction due to defective materials or construction. **THIS WARRANTY IS VOID IF THE HARDWARE HAS BEEN DAMAGED BY ACCIDENT, UNREASONABLE USE, NEGLECT, IMPROPER SERVICE, OR OTHER CAUSES NOT ARISING OUT OF DEFECTS IN MATERIALS OR WORKMANSHIP.**

WARRANTY DURATION

The Hardware is warranted for a period of three months from the date of the original purchase by the consumer.

WARRANTY DISCLAIMERS

ANY IMPLIED WARRANTIES ARISING OUT OF THIS SALE, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, ARE LIMITED IN DURATION TO THE ABOVE THREE-MONTH PERIOD. TEXAS INSTRUMENTS SHALL NOT BE LIABLE FOR LOSS OF USE OF THE HARDWARE OR OTHER INCIDENTAL OR CONSEQUENTIAL COSTS, EXPENSES OR DAMAGES INCURRED BY THE CONSUMER OR ANY OTHER USER.

Some states do not allow the exclusion or limitation of implied warranties or consequential damages, so the above limitations or exclusions may not apply to you in those states.

LEGAL REMEDIES

This warranty gives you specific legal rights, and you may also have other rights that vary from state to state.

PERFORMANCE BY TI UNDER WARRANTY

During the above three-month warranty period, defective Hardware will be replaced when it is returned postage prepaid to a Texas Instruments Service Facility listed below. The replacement Hardware will be warranted for three months from date of replacement. Other than the postage requirement, no charge will be made for replacement.

TI strongly recommends that you insure the Hardware for value prior to mailing.

TEXAS INSTRUMENTS CONSUMER SERVICE FACILITIES

U.S. Residents

Texas Instruments Service Facility
P.O. Box 2500
Lubbock, Texas 79408

Canadian Residents only

Geophysical Services Incorporated
41 Shelley Road
Richmond Hill, Ontario, Canada L4C5G4

Consumers in California and Oregon may contact the following Texas Instruments offices for additional assistance or information.

Texas Instruments Consumer Service
831 South Douglas Street
El Segundo, California 90245
(213) 973-1803

Texas Instruments Consumer Service
6700 Southwest 105th
Kristin Square, Suite 110
Beaverton, Oregon 97005
(503) 643-6758

IMPORTANT NOTICE OF DISCLAIMER REGARDING THE PROGRAMS

The following should be read and understood before purchasing and/or using the software media.

TI does not warrant that the Programs will be free from error or will meet the specific requirements of the consumer. The consumer assumes complete responsibility for any decision made or actions taken based on information obtained using the Programs. Any statements made concerning the utility of the Programs are not to be construed as expressed or implied warranties.

TEXAS INSTRUMENTS MAKES NO WARRANTY, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY IMPLIED WARRANTIES OR MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, REGARDING THE PROGRAMS AND MAKES ALL PROGRAMS AVAILABLE SOLELY ON AN "AS IS" BASIS.

IN NO EVENT SHALL TEXAS INSTRUMENTS BE LIABLE TO ANYONE FOR SPECIAL, COLLATERAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES IN CONNECTION WITH OR ARISING OUT OF THE PURCHASE OR USE OF THE PROGRAMS AND THE SOLE AND EXCLUSIVE LIABILITY OF TEXAS INSTRUMENTS, REGARDLESS OF THE FORM OF ACTION, SHALL NOT EXCEED THE PURCHASE PRICE OF THE SOFTWARE MEDIA. MOREOVER, TEXAS INSTRUMENTS SHALL NOT BE LIABLE FOR ANY CLAIM OF ANY KIND WHATSOEVER BY ANY OTHER PARTY AGAINST THE USER OF THE PROGRAMS.

Some states do not allow the exclusion or limitation of implied warranties or consequential damages, so the above limitations or exclusions may not apply to you in those states.

ADDENDUM

Editor/Assembler Owner's Manual

Please mark the following changes in your manual.

Page	*Section*	*Description*
42	3.1.3.1	In the last sentence of the first paragraph, change "least" to "most".
92	6.10	In the second line of the example explanation, change "value of ADDR" to "value in ADDR".
104	6.14.2	In the example, change "MOV *11,1" to "MOV *11+,1".
127	7.20.1	In the next-to-last line, change ">2220" to ">C220".
168	10.5	In the example, change ">2A41" to "@>2A41" and "Register 3" to "Register 2".
262	16.2.4	(Add the following.) NOTE: Some devices modify the GROM read address. RS232 and TP are known offenders. If your program accesses these devices, save the current GROM address (see section 16.5.2) before the I/O operation, and restore it (see section 16.5.1) after your program has accessed the device. Otherwise, the program will not be able to return to the Editor/Assembler or BASIC or to perform a BLWP @GPLLNK properly.
289	17.2.6	Change line 130 in the BASIC program to CALL LOAD("DSK1.BSCSUP","DSK2.STRINGO") This assumes that you have entered the source file on the next page by means of the Editor, saved the file as DSK2.STRING, and run the Assembler, using DSK2.STRING for a source file and producing DSK2.STRINGO as an object file.
328	21.1	The default for VDP Register 7 is >07 in TI BASIC and Extended BASIC.
335	21.5.2	In the last paragraph, change ">00 or >04" to ">03 or >07".
335	21.5.3	In the last paragraph, change ">00 or >04" to ">7F or >FF".
415	24.4.8	Change the second instance of GRMRD to "GRMRA EQU >9802".
416	24.4.8	Change the second line to "NUMREF EQU >200C".
420	24.5	(Add the following.) NOTE: A program to be saved using the SAVE utility should not have an entry point defined on the END statement. If you want to save the Tombstone City game in memory image format, you must first change the last line from "END START" to "END" and then reassemble the program. Otherwise, the game starts to run as soon as it is loaded, and you will not have a chance to execute the SAVE utility.
465	Index	VDP Write-Only Registers: add page 267 to references.

1053577-2